



MangoSynth24: Web-based Synthesizer

Seán Óg Durack Monks

N00191671

Supervisor: Mohammed Cherbatji

Second Reader: Timm Jeschawitz

Year 4 2022-23

DL836 BSc (Hons) in Creative Computing

Declaration of Authorship

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

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

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

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

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

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

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

DECLARATION:

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

Student : Seán Óg Durack Monks

Signed



Seán Óg DM

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

Abstract

The aim of this project was to construct a web-based subtractive FM synthesizer utilizing the Web Audio API to provide an experience as close as possible to a traditional desktop-based software synthesizer. The rationale for this was to determine the feasibility of the web browser as a platform for music production and audio synthesis, as the internet becomes more and more of a platform for software which was traditionally built for native, desktop platforms. The purpose of the application is to enable the creation of unique and interesting sounds without any of the prerequisites typically needed to do so. The steps involved in the development of this application was split into five phases: research, requirements, design, implementation, and testing. Testing was carried out throughout and after implementation. Results from the testing show that the application is enjoyable to use for both beginner and experienced synthesizer users. Further work that could be carried out include expanding the synthesizers' sound design capabilities with more effects and options for modulation, as well as building a backend consisting of a server and database so that users could save their synthesizer configurations to the cloud, making them accessible on any computer, and allowing the sharing of configurations on a public community database.

Acknowledgements

Firstly, I would like to express my sincere gratitude to the entire staff of IADT, with special thanks to every one of my lecturers over the past four years. Their support and guidance have been invaluable to my completion of this course.

I would also like to give my warmest thanks to my project supervisor Mohammed Cherbatji who made this work possible. His guidance and advice were instrumental to the success of this project.

A huge thanks to all my dear friends, in particular Paul Doyle and Jake Black who shared the experiences with me over the past four years at IADT, providing me with lots of support, motivation, and encouragement throughout. Without them this journey would have been a lot more challenging.

Lastly, I would like to thank my family for their endless support, love, and understanding throughout my academic journey. Thank you, my mother Sadie, my father Seán, and especially my grandfather Derrig, without his financial support throughout my studies none of this would have been possible.

Table of Contents

Declaration of Authorship.....	1
Abstract.....	2
Acknowledgements.....	3
1 Introduction	8
1.1 Overall Aim.....	8
1.2 Application Area.....	8
1.3 Technologies	8
1.4 Project Management	9
1.5 Requirements.....	9
1.6 Design.....	9
1.7 Implementation	9
1.8 Testing.....	9
2 Research.....	10
2.1 Basic Acoustics	10
2.2 Synthesizer Architecture.....	10
2.2.1 Oscillators.....	10
2.2.2 Sine Wave.....	11
2.2.3 Square Wave	11
2.2.4 Sawtooth Wave.....	11
2.2.5 Triangle Wave	12
2.2.6 Filters.....	12
2.2.7 Low-pass filter (LPF)	12
2.2.8 High-pass filter (HPF)	12
2.2.9 Band-pass filter (BPF).....	12
2.2.10 Envelopes	13
2.2.11 Modulation.....	13
2.2.12 LFOs (Low Frequency Oscillators)	13
2.2.13 Subtractive Synthesis	13
2.2.14 FM Synthesis	13
2.3 Literature Review.....	14
2.3.1 Web Audio	14
2.3.2 Web Audio API in Action	17
2.3.3 Examination of Feasibility	19

2.3.4	Conclusion	21
3	Requirements.....	22
3.1	Requirements gathering	22
3.1.1	Similar applications	22
3.1.2	Survey.....	29
3.2	Requirements modelling.....	50
3.2.1	Personas	50
3.2.2	Functional requirements.....	52
3.2.3	Non-functional requirements	52
3.3	Feasibility	53
3.4	Conclusion.....	54
4	Design.....	55
4.1	Program Design.....	55
4.1.1	Technologies (V1).....	55
4.1.2	Structure (V1).....	57
4.1.3	Technologies (V2).....	59
4.1.4	Structure (V2).....	60
4.1.5	Programming Paradigms.....	61
4.1.6	Design Patterns	61
4.1.7	Application Architecture	62
4.1.8	Process design.....	63
4.2	User interface design	65
4.2.1	Paper Prototypes	66
4.2.2	Low Fidelity Prototype (Wireframe)	66
4.2.3	High-Fidelity Prototypes	67
4.2.4	Style guide.....	70
4.3	Conclusion.....	72
5	Implementation	73
5.1	Scrum Methodology.....	74
5.2	Development environment.....	75
5.2.1	WebStorm	75
5.2.2	Git.....	75
5.3	Sprint 1.....	76
5.3.1	Item 1: Literature Review & Extra Research	76
5.3.2	Item 1: Prototyping – Paper Prototypes	76
5.3.3	Item 2: Prototyping – Wireframes & Prototypes	76

5.4	Sprint 2.....	76
5.4.1	Item 1: Requirements Gathering	76
5.4.2	Item 3: Prototype Improvements	77
5.4.3	Item 4: Google Forms Survey.....	77
5.4.4	Item 5: Design Document	77
5.5	Sprint 3.....	77
5.5.1	Item 1: Initial Development – Application Structure	77
5.5.2	Item 2: Initial Development – React, Vite, Zustand, Immer	77
5.5.3	Item 3: Update of Previous Documents.....	78
5.5.4	Item 4: Implementation Chapter	78
5.6	Sprint 4.....	78
5.6.1	Item 1: Migrating to Vanilla	78
5.6.2	Item 2: Tone.js Logic	79
5.7	Sprint 5.....	80
5.7.1	Item 1: Toggleable Synth Groups.....	80
5.7.2	Item 2: FX Switching.....	81
5.7.3	Item 3: LFO Modulation	82
5.7.4	Item 4: Recorder, Missing Controls, Interface Update, MIDI	82
5.7.5	Item 5: Hosting.....	83
5.7.6	Item 6: Interim Presentation.....	84
5.7.7	Item 7: MS24 Help Screen.....	84
5.7.8	Item 8: Synth Preset Functionality.....	84
5.7.9	Item 9: Code Clean-up	85
5.8	Sprint 6.....	85
5.8.1	Item 1: FM, AM, and Unison controls	86
5.8.2	Item 2: Oscillator Visualisations: Oscilloscopes	91
5.8.3	Item 3: LocalStorage Presets List	95
5.8.4	Item 4: Lossless Recording	96
5.8.5	Item 5: Electron Experiment	102
5.8.6	Item 6: User Testing	104
5.9	Sprint 7.....	104
5.9.1	Item 1: Keyboard Logic Rework	104
5.9.2	Item 2: GUI Rework.....	110
5.9.3	Item 3: Randomization.....	120
5.9.4	Item 4: Tooltips	122
5.10	Sprint 8.....	126

5.10.1	Finishing Touches	126
5.11	Conclusion	127
6	Testing	128
6.1	Functional Testing	128
6.1.1	User Input	128
6.1.2	Synthesizer Functionality	129
6.1.3	Discussion of Functional Testing Results	130
6.2	User Testing	130
6.3	Conclusion	135
7	Project Management	136
7.1	Project Management Tools	136
7.1.1	Notion	136
7.1.2	Git & GitHub	137
7.1.3	GitKraken	138
7.1.4	WebStorm TODO Comments	138
7.2	SCRUM Methodology	139
8	Project Reflection	140
8.1	Your views on the project	140
8.2	Completing a large software development project	140
8.3	Working with a supervisor	141
8.4	Technical skills	141
8.5	Further competencies and skills	141
8.6	Potential further developments	142
9	Conclusion	144
10	Bibliography	145
11	Annotated Bibliography (Literature Review)	146
12	Appendices	151

1 Introduction

1.1 Overall Aim

The overall aim of this project was to design and develop a web-based subtractive FM synthesizer using the Web Audio API, with the purpose of researching the feasibility of the web browser as a platform for music production and audio synthesis. Another aim of the project was to provide an online tool capable of generating many sounds, which could serve as both an introduction to audio-synthesis for people with no synthesizer experience, and as an accessible sound design and music production tool for experienced synthesizer users.

1.2 Application Area

The application area of this project was web-based audio technology, with a particular focus on audio synthesis techniques such as subtractive and frequency modulation (FM) synthesis. Advanced web-based audio technology is relatively new, with the Web Audio API, an interface which vastly expanded the audio capabilities of the browser having been created in 2011, which received an official recommendation from the World Wide Web Consortium in 2021.

1.3 Technologies

The development of the synthesizer involved the use of various technologies that were chosen for their suitability for web-based audio synthesis and creating a user-friendly and responsive web application.

Vanilla JavaScript was used to implement the main logic of the synthesizer, providing the ability to interact with the user interface and create the main audio logic.

HTML was used in combination with the `webaudio-controls` library to structure the user interface of the synthesizer, providing a foundation for the design and layout of the interface, with the library providing many Graphical User Interface (GUI) controls that mimic the controls found on hardware synthesizers.

TailwindCSS is a CSS framework which was used to style the synthesizer's user interface. It was chosen for its ease of use, flexibility, and ability to speed up the design process.

Tone.js is a Web Audio library which was used to create the main audio processing capabilities of the synthesizer. It was chosen for its premade transport system, audio effects, and event management capabilities, which sped up development times tremendously.

Vite is a build tool which was used to compile and bundle the synthesizer's code into an optimized package for deployment to the web. It was chosen for its speed and ease of use, providing a quick and efficient way to bundle the code and prepare it for deployment.

1.4 Project Management

The development of the synthesizer was carried out in a structured manner, by following the SCRUM methodology and through use of project management tools such as Notion, GitHub, and GitKraken. These tools were used to organise tasks, track progress, and for source code version control.

1.5 Requirements

The requirements phase involved identifying the necessary features for the synthesizer to meet the project's goals. This consisted of examining similar applications, as well as conducting a survey so that users needs and expectations could be met by the synthesizer.

1.6 Design

The design phase of the project involved deciding on the program design and user interface design for the synthesizer. The technical requirements necessary for the synthesizer to perform optimally were first examined, before a user-friendly and inviting user interface was planned, so that the application would appeal to as large an audience as possible.

1.7 Implementation

The implementation phase of the project involved developing the synthesizer using the technologies and requirements identified in the previous phases. The development process involved writing and testing the code, with supervisor meetings taking place every week and goals being set every two weeks in accordance with the principles of SCRUM.

1.8 Testing

Testing was a crucial aspect of the project. It was not only carried out throughout but also after implementation. The testing phase involved conducting functional testing and several usability tests to help with identifying and resolving bugs, optimizing the synthesizer's performance, and ensuring that the synthesizer met the project's requirements. The results of the testing showed that the synthesizer was an enjoyable and usable tool for both beginner and experienced synthesizer users.

2 Research

In this chapter, a brief introduction to acoustics and synthesizer architecture will be given to provide context around the physics of sound and the main components of tools commonly used to create sounds in electronic music. Following this, the main options music producers and sound designers have at their disposal will be described, such as analogue synthesizers, virtual synthesizers, and digital audio workstations (DAWs). The end of the chapter will consist of a literature review, written to determine *The Feasibility of the Web Browser as a Platform for Interactive Audio Synthesis*, with a focus on the Web Audio API.

2.1 Basic Acoustics

To understand how synthesizers can create a wide variety of different sounds, one must first understand the basics of sound itself.

When anything vibrates, the air molecules in the surrounding area begin to vibrate as well. These vibrating air molecules travel very similarly to how ripples in water travel after a pebble is thrown in; propagating from the source, the strength of the ripples lessening the further they travel. When vibrating air molecules or “sound waves” enter your inner ear, the brain perceives these vibrations as sound. What determines the pitch of the sounds we hear is the *frequency* (speed) at which an object *oscillates* (vibrates), which we measure in Hertz (Hz). The frequency of a sound determines the number of rarefactions and compressions, or “cycles” that are completed every second. If a vibrating object completes 100 cycles per second, it has a frequency of 100Hz. (Snoman, 2019, pp. 47-48)

Low frequencies like 100Hz are perceived as a low hum, known as bass, which can be felt in the body when played through a loudspeaker. Humans, on average, can perceive any frequency between 20Hz and 20,000Hz (20kHz), although most people lose the ability to hear a few thousand of the topmost frequencies throughout their lives. (Farnell, 2010, p. 78)

2.2 Synthesizer Architecture

Synthesizers have revolutionized the music industry by providing musicians with a vast palette of sounds, available to be mixed and manipulated in a variety of ways to create unique sounds and musical compositions. Understanding the architecture of synthesizers is essential for anyone who wants to program them effectively and make music that stands out in today's competitive landscape. This subsection of the thesis will explore the various components that make up a synthesizer's architecture, including oscillators, filters, envelopes, and modulation sources, and discuss how they can be combined and configured to achieve specific sonic goals.

2.2.1 Oscillators

Oscillators are at the core of sound synthesis in electronic music. They generate periodic waveforms (such as a sine, square, sawtooth, and triangle waves) at various frequencies, to produce the fundamental tones of the instrument. These waveforms can then be manipulated by other parts of the synthesizer to add harmonics and variety.

2.2.2 Sine Wave

The sine wave is the simplest of the waveforms. It is often thought of as “pure”, as there are no extra harmonics, all you hear is the frequency it oscillates at. It’s a smooth sound to the ears, as there are quite literally no edges to the waveform. It is often used to recreate gentle, soothing instruments such as flute or bowed string instruments.

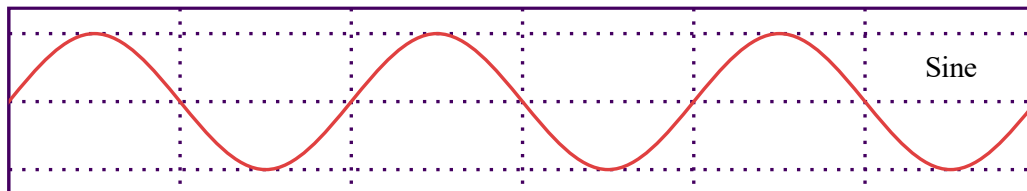


Figure 1. Sine Wave

2.2.3 Square Wave

The square wave is another very simple waveform, especially for analogue synthesizers, since it only exists in two states: high and low. As a result of the sharp rise and fall, this wave has a harsh, abrasive sound when compared to the soft sine. It also exclusively produces odd harmonics, which makes it sound a bit hollow. The square wave is often used to create punchy percussive sounds, as well as wide, crunchy bass sounds. It is probably best known for its use in classic video games and chiptune music.

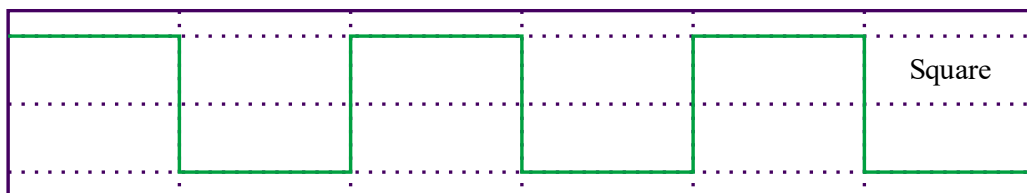


Figure 2. Square Wave

2.2.4 Sawtooth Wave

The sawtooth wave is characterized by a ramp-like shape that resembles the teeth of a saw. It produces all the even and odd harmonics in the series, and as a result it is often used in electronic music to create bright, cutting sounds such as leads or pads. It is also a common waveform used for creating raspy bass sounds, especially in subtractive synthesis, since filtering such harmonically rich sounds can result in wonderful sweeping effects.

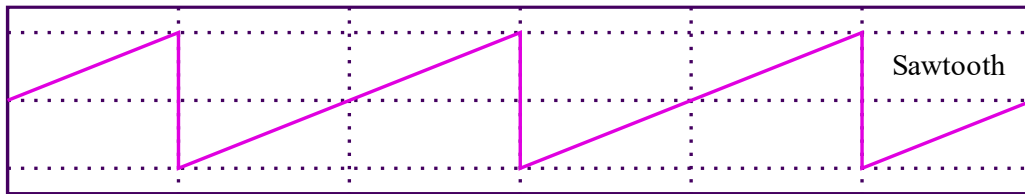


Figure 3. Sawtooth Wave

2.2.5 Triangle Wave

The triangle wave is like the sawtooth wave, but has a more mellow, rounded sound due to it only producing odd harmonics. It is often used to add a bit of brightness to mellow lead or pad sounds and is often as a modulation source for other waveforms.

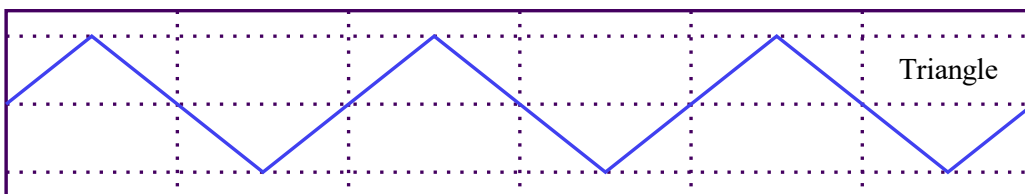


Figure 4. Triangle Wave

2.2.6 Filters

Filters are a fundamental component of synthesizers and other electronic instruments used in music production. Filters are used to shape the tone of a sound by removing or attenuating certain frequencies. Common types of filters include low-pass, high-pass, and band-pass filters.

2.2.7 Low-pass filter (LPF)

Attenuates frequencies above a certain 'cutoff' point and allows frequencies below that point to pass through unaffected. This type of filter is commonly used to create mellow, warm sounds by reducing the high-frequency content of a sound.

2.2.8 High-pass filter (HPF)

Attenuates frequencies below a certain 'cutoff' point and allows frequencies above that point to pass through unaffected. This type of filter is commonly used to create bright, cutting sounds by reducing the low-frequency content of a sound.

2.2.9 Band-pass filter (BPF)

Attenuates frequencies outside a certain range and allows frequencies within that range to pass through unaffected. This type of filter is commonly used to create unique sounds by emphasizing a specific range of frequencies.

2.2.10 Envelopes

Envelopes are used to control how a sound evolves over time. They typically consist of four parts: attack, decay, sustain, and release. Attack controls how quickly the sound reaches its maximum amplitude, decay controls how quickly the sound drops from its maximum amplitude to the sustain level, sustain controls the amplitude the sound holds for as long as the note is held, and release controls how quickly the sound fades away after the note is released.

2.2.11 Modulation

Modulation refers to the process of altering one or more parameters of a sound wave over time. In music production, modulation can be achieved using a variety of techniques, including amplitude modulation (AM), frequency modulation (FM), ring modulation, and phase modulation. These techniques can be used to create a wide range of effects, from subtle variations in timbre to extreme distortion and dissonance.

2.2.12 LFOs (Low Frequency Oscillators)

Low-frequency oscillators (LFOs) are a common tool used for modulation in electronic music production. LFOs generate a repeating waveform with a frequency below the audible range (typically between 0.1 Hz and 20 Hz), which is then used to modulate other parameters of a sound wave. For example, an LFO can be used to modulate the frequency of an oscillator, creating a vibrato effect, or to modulate the filter 'cutoff' frequency, creating a wah-wah effect.

2.2.13 Subtractive Synthesis

Subtractive synthesis is a method of sound synthesis that starts with a complex waveform, such as a sawtooth or square wave, and removes certain frequency components to create a desired tone. Subtractive synthesis is often used to create warm, analogue-sounding tones.

2.2.14 FM Synthesis

FM synthesis is a method of sound synthesis that uses one oscillator (the carrier) to modulate the frequency of another oscillator (the modulator). This creates a wide variety of complex and evolving sounds and is often used to create bell-like and percussive sounds. (*Snoman, 2019, pp. 55-73*)

2.3 Literature Review

'Feasibility of the Web Browser as a Platform for Interactive Audio Synthesis'

The web browser has become a platform for a wide range of applications, including but not limited to audio and video streaming, online gaming, social networking, e-commerce, and productivity tools. With the increase of capabilities of web browsers and the proliferation of high-speed internet access, it is now possible to use the web browser as a platform for many tasks that were previously only possible on dedicated software. This has led to the development of web-based alternatives to traditional desktop applications, as well as the creation of entirely new categories of applications that are only possible on the web. As a result, the web browser has become a central part of the modern computing experience for many people.

As web browsers continue to gain more capabilities and support for advanced features, it is worth examining the feasibility of using the web browser as a platform for music production and audio synthesis. The following literature review explores work done by other researchers, to evaluate the current state of the web browser as a platform for audio synthesis, including its capabilities and limitations, as well as the potential benefits and challenges of using the web browser for this purpose.

2.3.1 Web Audio

The term *web audio* refers to the ability to play and manipulate audio within a web browser using languages like HTML and JavaScript. With the increasing use of the web for streaming music and podcasts, and in recent years even for producing music itself, the ability to control and customize audio within web applications has become a crucial aspect of modern web development. In this section, we will explore the various tools and techniques available for working with audio on the web, including the HTML5 `<audio>` element, and the Web Audio API.

A Brief History

Smus (2013) explains that the original method of playing sounds on the web was through the `<bgsound>` HTML element, which allowed website authors to play background music for their website's visitors when a page was opened. This early implementation of audio in the browser was exclusive to Internet Explorer and was never standardized or introduced to other browsers, however, Netscape did implement a similar feature with the `<embed>` element, which provided "basically equivalent functionality".

Adobe's Flash was the first way to play audio on the web that was available no matter what browser you were using, the only drawback being that it required a browser plugin to run. Smus goes on to explain that more recently, HTML5's `<audio>` element became the norm across browsers, as it "provides native support for audio playback in all modern browsers". According to the author, the main problem with the `<audio>` tag is that it is severely limiting when faced with more advanced use-cases than basic audio playback, such as in video games and interactive web apps. Smus then gives examples of a few of these limitations, such as "no precise timing controls", "very low limit for the number of sounds played at once" and "no ability to apply real-time effects".

Following this, Smus gives an example of one of the “several attempts to create a powerful audio API on the Web to address some of the limitations” previously described. This example is the *Audio Data API*, developed by Mozilla for the Firefox browser, which started as an extension of the `<audio>` element’s JavaScript API. However, due to the “limited audio graph” presented by this API, it was never adopted, and was later deprecated in favour of the Web Audio API, the draft of which was originally proposed by Google. (Roberts, 2015)

Web Audio API

Smus describes the Web Audio API as a “high-level JavaScript API for processing and synthesizing audio in web applications”, which is “completely separate from the `<audio>` tag”. The goal of the API being to bring capabilities from modern game engines and native audio production suites to the browser. The result of this is a “versatile API” which can be used across a broad spectrum of audio-related tasks, from generating sound effects for web-based video games, to advanced interactive music synthesis applications modelling analogue and digital synthesizers.

Since its promising beginnings in 2013, the Web Audio API has come a long way. Choi (2018) presents the new *AudioWorklet* interface in his paper in which he discusses the future of the API by comparing the *AudioWorklet* to its predecessor, the *ScriptProcessorNode*.

According to Choi, the *ScriptProcessorNode* was originally the World Wide Web Consortium (W3C) Audio Working Group’s response to a high demand for more extensibility within the Web Audio API. It was the only component of the API which allowed the authoring of fully custom JavaScript code, providing a lot more control to developers. However, it “failed to meet the expectations of the developer community”, mainly due to all its computations being done on the browser’s main thread, as shown in Figure 5. This led to serious overcrowding in the thread’s task queue, meaning that important audio processes were very likely to be delayed when CPU intensive audio code was executed, leading to both audio and UI glitches.

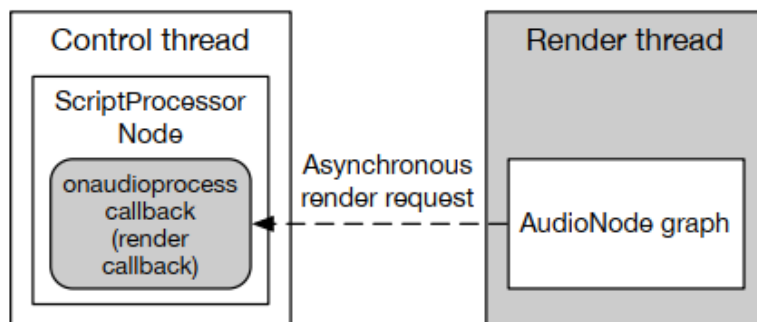


Figure 5. *ScriptProcessorNode* rendering mode (Choi, 2018)

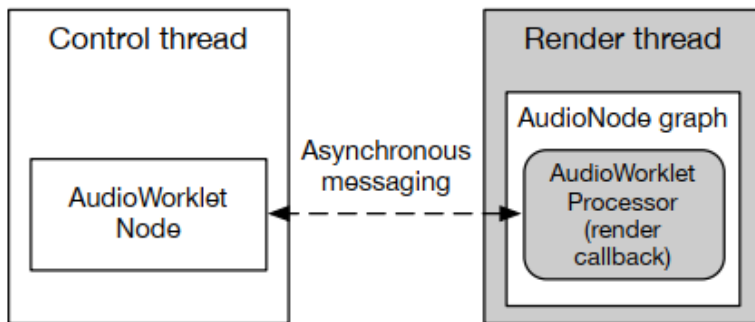


Figure 6. AudioWorklet node rendering mode (Choi, 2018)

The intention of the AudioWorklet was to address these issues, which it did in three ways.

Firstly, by cleanly separating the node and processor between the control thread (the browser’s main thread, primarily reserved for DOM manipulation and JavaScript code execution) and the render thread (the thread where the browser’s audio engine runs), less pressure was put on the control thread, and rendering stability and performance were improved significantly. This separation can be seen in Figure 6.

Secondly, synchronous rendering was employed, replacing the double buffering method which the ScriptProcessorNode used in Chromium browsers. This eliminated several issues, such as duplicated audio and audio dropouts.

Lastly, while ScriptProcessorNode was fully customizable, it was “far from being a first-class object”. To provide more extensibility, every aspect of the AudioWorklet’s operation, from class definition to processing, was made extendable in a way that was compatible with the API’s built-in AudioNodes.

The AudioWorklet was a crucial improvement to the Web Audio API, narrowing the divide between desktop music production software and the browser significantly. As Choi puts it, its release served as “an open invitation to the computer music community coming from the very wide world of web applications”.

As a more recent example of the progress made with the API over the years, in a press release from W3C in 2021 (W3C being an international organization that develops and maintains standards for the World Wide Web), the Web Audio API officially received the endorsement of the W3C membership, marking the specification as a *W3C Recommendation*. This new status signified that after ten years of development, the Web Audio API had reached the level of stability and dependability necessary for it to be used as a standard on the web.

Several well-respected W3C members and Industry Users gave testimonials under the press release, such as Honchan Choi, Google’s Web Audio Technical Lead, who said “Immersive audio experiences entertain and delight after just following a link on the web . . . Google is pleased to have supported Web Audio standardization and advocacy over this decade-long journey”, and Paul Adenot from Mozilla who stated “With the addition of AudioWorklet’s direct audio sample manipulation, the Web Audio API is now a viable foundation for a broad range of use-cases and applications”.

With this significant milestone achieved, the Web Audio API specification was given the credibility required for more developers around the world to confidently adopt it for use on their audio-related projects.

2.3.2 Web Audio API in Action

In this section of the literature review, we will examine the various ways in which the Web Audio API has been applied in practice, with a focus on existing research. By exploring the current state of the art in Web Audio API development, we aim to gain a better understanding of its capabilities and limitations, to determine the feasibility of the browser as an audio synthesis platform, as well as to identify areas for future research and development.

Web Applications

A myriad of web applications built using the Web Audio API can be found online today, ranging from modular sound synthesis environments like the [Web Audio Playground](#) to subtractive synthesizers like [Microtonal](#) from WebSynths, to fully-featured Digital Audio Workstations (DAWs) like the open-source [GridSound](#), or Spotify's collaborative DAW [Soundtrap](#).

As stated by Lind (2017), Soundtrap was “built using the latest Web Audio, Web MIDI, and WebRTC standards”, allowing for quick and easy collaborative music production across the world. Boasting over 1.5 million users by 2017, it's clear that there is a place for the DAW on the web. Soundtrap works on a variety of low-end hardware too, from laptops to smartphones (both Android and iOS), which it does by “auto-detecting basic performance characteristics on startup and during studio sessions”, before modifying the Web Audio graph to optimise for that device.

Many of these music-based web apps perform perfectly well as music production and sound design tools, allowing people around the world to express themselves musically by simply visiting a website on their computer or smartphone.

Libraries and Frameworks

Just as there are many standalone web-apps implementing the Web Audio API, there are also many libraries and frameworks built using it, each providing their own optimized suites of premade synthesizers and effects for developers to make use of, as well as important event scheduling functionality such as transport systems, to ensure precise synchronisation of each sound.

One such library is [Gibberish.js](#), which describes itself as a fast “JavaScript DSP library that creates JIT optimized audio callbacks using code generation techniques”. According to Roberts et al. (2015), the primary reason for JavaScript having excellent performance in the browser is due to the use of “just-in-time” (JIT) compilation. “The virtual machine detects the most heavily used functions, paths, and type specializations of the code as it runs, and replaces them with translations to native machine code”. While developing Gibber, a live-coding environment focused on sound synthesis, Roberts and his team found that the libraries built for the ScriptProcessorNode which were available to them at the time, were “not efficient enough to realize the complex synthesis graphs we envisioned”. This led them to start working on their own library, Gibberish.

Roberts et al. were successful in their attempts to create an optimized DSP library suitable for the Gibber project. By integrating Gibberish.js with a graphical user interface (GUI) library called Interface.js, they were able to create their live-coding environment the way they intended, in which a user writes JavaScript audio code with Gibberish syntax directly into the browser, while audio and

interactive graphics are dynamically rendered alongside the code, allowing the authoring, playback, and manipulation of synthesized sounds all from the one webpage. It is worth mentioning that Gibber and Gibberish are still receiving regular updates on GitHub to this day, Gibber 2 having released in 2021.



```
gibber      intro
share gabber restart engine  reference source code discord twitter
// hit alt+enter to run all code
// or run line/selection with ctrl+enter.
// ctrl+period to stop all sounds.

Theory.tuning = 'slendro'
Theory.mode = null

verb = Reverb( 'space' ).bus()
delay = Delay( '1/3' ).bus().connect( verb, .1 )

perc = FM[3]( 'perc' )
  .connect( delay, .65 ).connect( verb, .35 )
  .spread( .975 )
  .note.seq( sine( btof(8,7,0) ), 1/8, 0 )
  .note.seq( sine( btof(4,3,0) ), 1/16, 1 )
  .note.seq( sine( btof(8,7,7) ), 1/6, 2 )
  .loudness.seq( sine(4.33, .35, .7) )

kik = Kick()
  .trigger.seq( 1, 1/4 )

hat = Hat( { decay: .0125 } )
  .trigger.seq( [1, .5], 1/4, 0, 1/8 )

bass = Synth( 'bass.hollow' )
  .note.seq( [0, 1, 2, -1], 1 )
  .trigger.seq( [.75, .5, .25], [1/4, 1/8] )
```

Figure 7. A screenshot of Gibber 2's [playground](#) interface (2023)

Another example of Web Audio libraries and frameworks would be [Tone.js](#), a framework which facilitates the creation of interactive music applications in the browser. Tone offers a multitude of prebuilt synths and effects, as well as several DAW features such as a global transport system.

Mann (2015) explains that the development of Tone.js “was guided by three tenets: **musicality** . . . **modularity** . . . and **synchronisation**”. *Musicality* referring to the ability to define both scores and synthesizer architectures as JSON objects while also using “note names and rhythmic notation”, *modularity* referring to the availability of a wide range of building blocks for the synthesis and processing of audio signals, and *synchronisation* referring to the ability to match these building blocks and various audio events on a shared timeline.

After explaining various components of the high- and low-level architecture of Tone.js, Mann compares the library to other similar libraries like WAAX, Gibberish, and Lissajous, stating that Tone distinguishes itself in three ways. To sum it up, firstly Tone simplifies the development process by accurately synchronising multiple *AudioParams* (parameters controlling different high-level components such as an oscillator), leading to less code repetition compared to other libraries. An example of this is that to set the frequency of a synthesizer in WAAX, three separate parameters need to be adjusted to achieve the same effect as changing a single parameter in Tone. The second example Mann gives for how Tone distinguishes itself is that it is very compatible with other libraries and modules, especially compared to Gibberish which does all its audio processing in a single *ScriptProcessorNode*, which makes it difficult to route its audio through other components. Lastly, in Tone.js, event scheduling is done by using JavaScript callbacks, which adds a lot of flexibility for sequencing and scheduling events. Lissajous on the other hand, can only handle loop-based events

due to how it abstracts away sequencer callbacks, restricting users to only scheduling specific events like note triggering. These features “give Tone.js the flexibility to create a wide range of music”.

2.3.3 Examination of Feasibility

Two papers were found which closely related to the question of how feasible the web browser is for audio synthesis and computer music production. The first of these papers examined the suitability of the web browser as a computer music platform, while the other evaluated the Web Audio API in terms of creating a virtual analogue synthesizer.

As a Computer Music Platform

The first of these papers, titled *The Viability of the Web Browser as a Computer Music Platform* from Wyse and Subramanian (2013), very closely match the topic of this paper. The authors ask, “Why would musicians care about working in the browser, a platform not specifically designed for computer music?”, giving Max/MSP as an example of a platform which *is* specifically designed for computer music. One possible reason they give is that the browser offers attractive alternatives for musicians, such as the extreme versatility of the medium, as it is one shared by a huge community of developers and provides an immense selection of libraries for a near-endless number of applications, “from physics and graphics to user interface components, specialized mathematics, and many other areas relevant to computer music developers”. Another advantage given is that with such large open-source communities, the infrastructure for participation on large, complex projects, is already well developed, meaning there is a higher chance of finding others to collaborate with and get support from when developing an app.

Wyse and Subramanian go on to say that there are also benefits for the “users” of these web apps, from “performers and actively participating audiences” to “composers building on sounds or instrument designs contributed by others”. An example of the former given is William Duckworth’s *Cathedral* from 1997, being “one of the earliest examples of interactive music and graphical art on the Web”. Roberts and Kuchera-Morin (2013) give further examples of this, with Gibber being used in several live performances since its introduction, including a formal concert performance by the CREATE Ensemble at UC Santa Barbara, “in which six performers submitted code to a remote computer for execution”, none of which had any programming experience but were still able to participate by “copy and pasting code and modifying variable values”. Wyse and Subramanian state that the Web Audio API “is one of the new APIs that is of critical importance for computer music”, explaining that major browsers already supported a wide range of computer music applications in mid-2013, giving examples like Gibber, and “multi-player shared sequencers and instruments” from Patrick Borgeat. The authors conclude that the Web Audio API is a key enabler of web-based music production, which is closing the gap between the browser and native platforms. There are still some areas in which that gap is still open, but it “appears to be closing”. Wyse and Subramanian say they “expect the growth in capabilities to continue apace”, with the unique capabilities of the browser already offering new creative possibilities for musicians and programmers to explore.

As a Virtual Analogue Synthesizer

In this inductive thesis, Eriksson (2013) evaluates “the suitability of the Web Audio API to implement virtual analog synthesizers”, firstly by defining a reference architecture for analogue synthesizers, and then by creating a categorization and point system for evaluating the API with.

To define a reference architecture, Eriksson examined six synthesizers, both analogue and virtual analogue, using the official manuals and similar references to define three levels of specifics for their own reference.

At the first level, the “top level”, different sections of the synthesizers they examined were modelled based on how they interact with each other, using block diagrams for each synth and then comparing common units in a single block diagram overlay, giving an overview of what a synthesizer architecture looks like.

At the second level, the “section level”, sections from the top-level block diagram were defined in design terms and by what elements made up those sections. This was also modelled using block diagrams and aimed to provide design suggestions for each section of the synthesizer.

At the third level, or “element level”, common properties of the elements that make up the individual sections from level two are defined, by creating a matrix of properties, sorted into three levels of necessity; required, recommended, and optional. Required properties were found in over 90% of all the synthesizers examined, while recommended properties were found in 60-90% of them, and optional properties were only found in 30-60% of them. By defining these levels of necessity, it would be possible to determine what to expect in the elements from level two of the architecture.

When it came to evaluating the Web Audio API, three levels of achievability were defined: native, implementable, and non-native. Native properties being readily available in the API, implementable properties being easily achieved with custom JavaScript code through the API’s customizable nodes, and non-native properties being very difficult to mimic without writing a lot of complex code.

The result of Eriksson’s evaluation was a score of approximately 80%, with 5 out of 5 possible points scored at the top level, 8 out of 9 scored at the section level, and 44 out of 68 scored at the element level; with 22 out of 23 required elements, 9 out of 14 recommended elements, and 23 out of 31 optional elements being achievable. Eriksson concludes that this score suggests that the API can handle most behaviours required to implement the reference architecture and is happy with the API’s performance in all areas but the modulation section, mostly due to the rigid implementation of the OscillatorNode, which does not allow direct access to properties like the phase of a waveform. The API also lacks an option for automating custom numeric properties natively. Without these downfalls, the score would likely have improved significantly.

Despite the areas for improvement, the resulting score of the evaluation is high enough to suggest that the Web Audio API was feasible for virtual analogue synthesizers in 2013, long before the API’s addition of the AudioWorklet node and its official recommendation from W3C.

Closing Thoughts

In the “expert commentary” at the end of the chapter titled *2013: The Web Browser as Synthesizer and Interface* authored by Roberts et al. from *A NIME Reader* (Jenselius et al. 2017), Abram Hindle gives his thoughts on how far we’ve come from the days before HTML5 and the Web Audio API,

stating “HTML5 lets us deploy a synthesizer on every desk and in every smartphone pocket. Mass adoption combined with portability makes any HTML5-enabled device a potential synthesizer or NIME” (i.e., New Interface for Musical Expression). Hindle explains that the work of Roberts et al. on the Gibber project made it clear to him that “WebAudio is here to stay”.

2.3.4 Conclusion

In conclusion, the literature reviewed in this paper demonstrates the versatility and potential of the Web Audio API as a tool for creating interactive audio experiences on the web. Through its various features and functions, the API has been used to build a wide range of applications, from synthesizers and digital audio workstations to audio effects processors and beyond. While the API has proven to be a powerful tool, the studies reviewed also highlight some areas where its capabilities could be improved or expanded upon. Future research could focus on these areas, as well as on the development of new and innovative uses for the API. In conclusion, the Web Audio API is a valuable resource for web developers looking to incorporate audio into their projects, and its continued evolution will likely lead to even more exciting and engaging audio experiences on the web in the future.

3 Requirements

The purpose of the requirements phase of a project is to allow for developers to work out what the application should be able to do. It is important to understand what the potential users of an application would like it to do rather than the developer deciding what is required.

3.1 Requirements gathering

The beginning of the requirements phase starts with the gathering of requirements. This is done by examining similar applications, and by querying potential users of the application, often through surveys and interviews. This is done so that educated decisions can be made during the following phase, the requirements modelling phase, in which the requirements for the project are set, so that design and implementation can run smoothly.

3.1.1 Similar applications

To start with, several similar applications were examined. The first similar applications looked at were traditional software synthesizers using the VST plugin interface. Following this, web-based software synthesizers using the Web Audio API were inspected.

3.1.1.1 *Diva* – “The spirit of analogue” (VST)

Diva is an award winning virtual analogue synthesizer plugin or VST (Virtual Sound Technology), developed by Berlin-based synthesizer manufacturer U-He with the goal of capturing “the spirit of five decades of analogue synthesizers”.



Figure 8. U-He's Diva

It features a semi-modular interface made up of many oscillators, filters, and modulators, as well as a flexible modulation matrix for complex sound design. Its user interface is friendly and features a classic look and feel which appeals to many electronic music producers.

Its semi-modular design allows for an extremely wide range of sounds to be made with it, ranging from accurate analogue replications to crisp, modern sounds that wouldn't have been possible on an old synthesizer.

Advantages

- Modelled on real analogue synthesizers.
- Can mix and match major panels (semi-modular).
- 5 oscillator models, 5 filter models, 3 envelope types.
- User-definable modulation sources & modulation matrix.
- Over 1200 presets (pre-loaded configurations) out of the box.
- It's ready to make sounds on start-up, all major components are configured.

Disadvantages

- Very CPU intensive.
- The user-interface could be intimidating to beginners.
- Could be seen as limiting compared to more modular synthesizers.

3.1.1.2 [Phase Plant](#) – “Build the synth of your dreams” (VST)

Phase Plant is another award-winning synthesizer VST, made by Swedish developers Kilohearts. While this synthesizer is also technically semi-modular, it takes a very different approach to Diva, providing its users with a lot more freedom to build patches (a term often used for the configurations of modular synthesizers) in almost any way they would like.

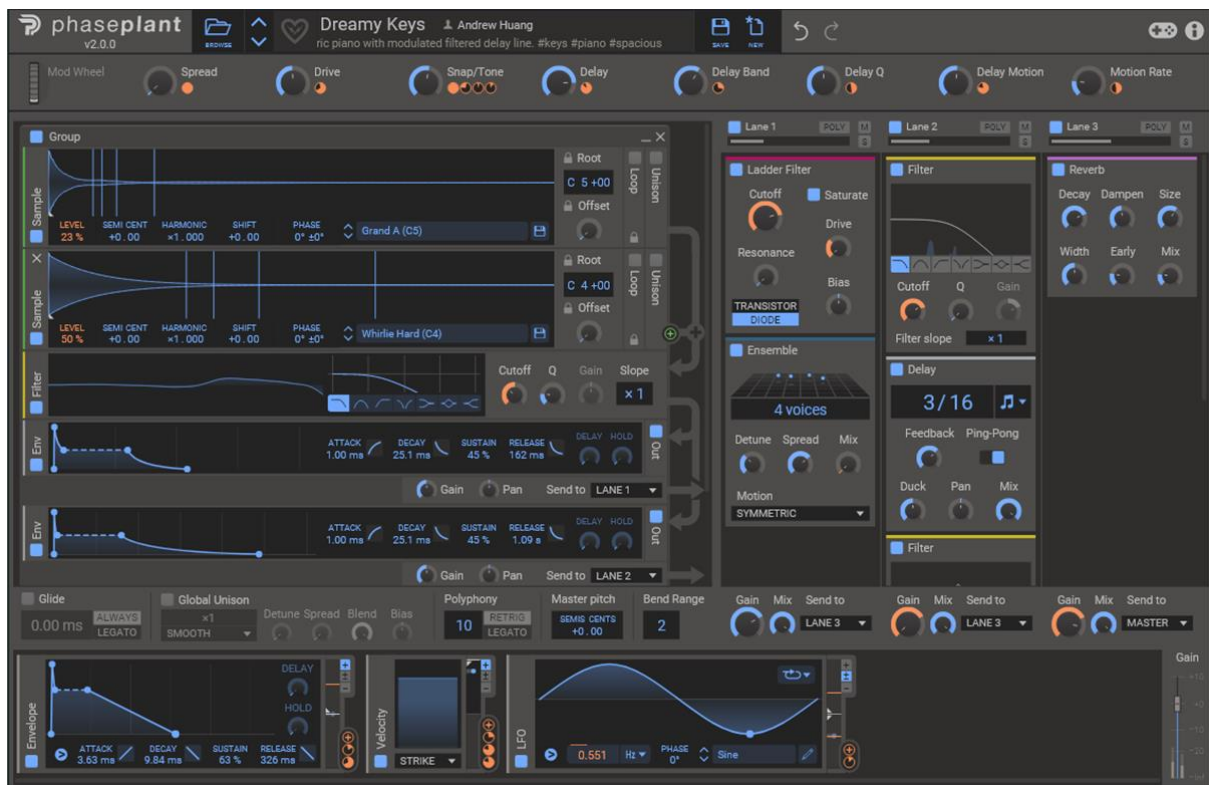


Figure 9. Kiloheart's Phase Plant

When you first start-up Phase Plant, you are presented with a blank interface consisting of the three main sections of the window, which can be seen in Figure 9: the generator section (middle-left), effects section (middle-right), and the modulation section (bottom). While this may seem daunting at first, it's surprisingly easy to get started with. To make a sound, one must only add a generator and send a MIDI signal. To add distortion to that sound, it is as simple as clicking one of the three lanes in the effects section and selecting one of the distortion effects. To add some movement to the distortion, so that it fades in and out over time, you would click in the modulation section, select an LFO (Low-Frequency Oscillator) and connect it to one of the distortion effect's parameters.

For users who are new to music production and sound design, or for those who just don't want to start from scratch, Phase Plant ships with over 300 presets offering a huge variety of sounds from sharp plucks to sweeping cinematic pads. These can also be used as a starting point for beginners who want to learn how to make their own sounds, by loading a preset and modifying it.

Advantages

- Almost fully modular architecture (only the most fundamental routing is unchangeable).
- Allows the combination of *any* number of generators, effects, and modulators.
- Almost limitless potential for sound design.

Disadvantages

- Complex patches can be CPU intensive.
- Could be very intimidating for beginners.

- Some effects must be purchased separately.

3.1.1.3 [webOBXD \(web-based\)](#)

webOBXD is a WAM (Web Audio Module) port of a JUCE plugin (for use in DAWs) called OBXD, which itself is an emulation of the famous OB-X, OB-Xa, and OB-8 analogue synthesizers from Oberheim. The WAM implementation was developed by Jari Kleimola in 2017, bringing it to the browser making it easily accessible to all.



Figure 10. WebOBXD

OBXD is a subtractive synthesizer (meaning that sounds are shaped by through use of filters) featuring two oscillators (saw and pulse shapes), a multi-mode filter, amplitude and filter envelopes, unison, voice spreading and variation, as well as simple LFO-based modulation. It is non-modular, meaning what you see is what you get, but what it provides is more than enough to make a myriad of classic, spacey, Oberheim-inspired sounds.

Advantages

- Simple, yet effective.
- Comes with many patches.
- Emulation of famous classic synthesizers.

Disadvantages

- User interface might seem cluttered or confusing to some.
- Non-modular architecture, less flexible for sound design.
- No inbuilt way to save patches or sounds created.
- Only features sawtooth and pulse wave shapes.

3.1.1.4 [Microtonal](#) (web-based)

Microtonal is a web-based semi-modular subtractive synthesizer developed by Mitch Wells as part of WebSynths, a small collection of free, hosted synthesizers and drum machines.

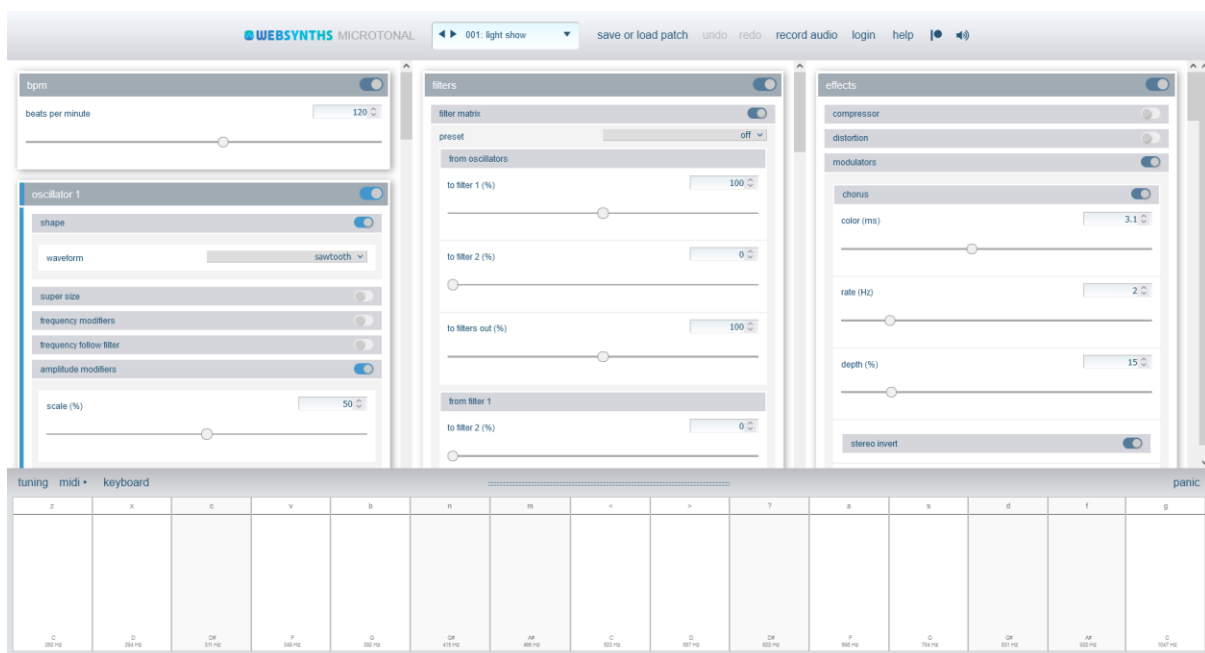


Figure 11. Microtonal

Microtonal features three oscillators, each with all major shapes available, 5-15 voice unison, frequency modulation, amplitude modulation, convolution, and an equalizer. It also features two multi-mode filters, an amplitude envelope, a master equaliser, limiter, and several effects, such as distortion, chorus, delay, and reverb.

Advantages

- Large selection of toggleable modules.
- Lots of parameters for each module.
- Can record audio when logged in.
- Somewhat responsive webpage.

Disadvantages

- UI only contains text, sliders, and toggles.
- Parts of the UI are always hidden no matter the screen size, scrolling is required.
- Lots of “menu diving” (seemingly never-ending depths of menus and submenus)

3.1.1.5 [Web Audio Playground](#) (web-based)

Web Audio Playground is a fully modular synthesis environment made by Chris Wilson which describes itself as “a demonstration of the web audio API”. It uses the React Flow library to provide an interactive node-based user interface, allowing you to add modules, move them around, and connect them to each other in whatever way you want.

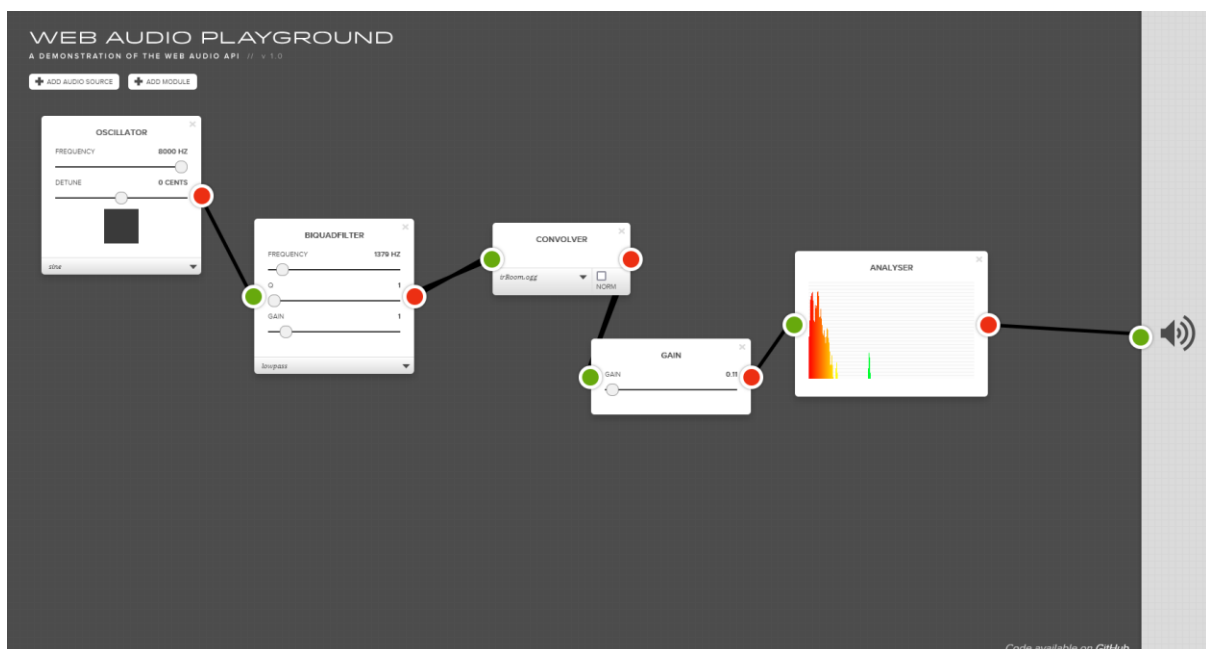


Figure 12. Web Audio Playground

Web Audio Playground provides three different types of audio source: audio samples, oscillators with all basic waveforms as options, and live audio device input. It also provides six different audio modules: a multi-mode filter, delay effect, dynamics compressor, gain controller, convolver, and a spectrum analyser. The means for controlling the audio are quite restrictive, as you are limited to using the mouse, and there aren't all that many audio modules, but it is nonetheless impressive to find a node-based modular approach on the web, which demonstrates the node-based architecture of the Web Audio API in a visual and interactive way.

Advantages

- Fully modular.
- Demonstrates the Web Audio APIs underlying architecture.

Disadvantages

- Not many options.
- No keyboard control.
- Extremely loud if not carefully set up (no master limiter).

3.1.1.6 Summary

In this section we looked at five software synthesizers, two of which were native (VSTs) and three of which were web-based, using the Web Audio API. These synthesizers ranged from non-modular to fully modular, and mainly used subtractive synthesis and frequency modulation as their main methods for providing the ability to create a wide range of different sounds.

Diva shows how a semi-modular approach can both lead its users towards a certain vintage sound while also being highly customisable; providing the ability to swap in and out several oscillator and filter groupings, while its signal flow being the same no matter what. Phase Plant on the other hand shows how a semi-modular approach can give the user a lot more freedom, choosing however many signal generators, effects, and modulators as they wish, with the only drawback being that without sufficient knowledge, it could be difficult to create polished-sounding patches, since it starts as a blank slate.

On the web side of things, webOBXD demonstrated a non-modular approach, where the only options provided are the tweaking of parameters, with no ability to interchange signal generators and providing no inbuilt effects. This approach is more traditional in the sense that the synthesizer has a certain “sound”, due to there being a more limited number of possible configurations. Microtonal was an example of a semi-modular web-based synthesizer, which provides a great many options for tweaking sounds, but falls short on the user interface side of things, looking less like a traditional synthesizer and more like a website with an abundance of toggleable menus and sliders to control parameters. Lastly, we looked at Web Audio Playground, an interactive node-based modular synthesizer which showcases the nodal architecture of the Web Audio API. In this environment, the user is given total control of the signal flow, connecting each audio module with virtual cables to create a sound. While this approach gives the most freedom, it requires an certain level of experience with synthesizers to know what should connect to what. Another issue is that the Web Audio Playground does not provide safeguards such as a master limiter, meaning it can create extremely loud sounds if set up incorrectly, which could damage a user’s hearing or their playback devices.

From this analysis, we can see that giving more control to the user isn’t always better when it comes to synthesizers. To appeal to as many users as possible, it is important to balance the modularity of a synthesizer with the direction it leads you in. With a non-modular synth, you are entirely led by the signal flow; beginning with an oscillator, a signal flows through an amplitude envelope into a filter, then into a filter envelope modulated by an LFO, before finally flowing through effects and being output. With an entirely modular synth, the user decides the flow, they are not led in any way, and thus it is possible to make a patch that produces no sounds, broken sounds, or, given the right modules and know-how, any sounds imaginable. Semi-modular approaches seem to be the happy medium, both leading users toward commonly desired sounds, while letting them choose from a

range of carefully selected modules to provide them with more means of self-expression than a non-modular synth would.

User interfaces are also important. Providing many tweakable parameters means having the challenge of trying to fit those parameters into a limited space, while trying to avoid a cluttered, confusing layout. Most of the examples shown are inspired by analogue synthesizer interfaces, but a lot of analogue synthesizers are well known to be confusing. Another challenge is where to put each grouping of parameters? A common theme seems to be left-to-right, matching where the signal begins and ends, but what varies a lot between synthesizers is vertical positioning. Without arrows, prior knowledge, or trial and error, it can be impossible to tell what the signal flow is, which could make things even more confusing to beginners.

3.1.2 Survey

To help with gathering requirements, a Google Forms survey was conducted. A link to this survey was shared to the class Discord server, as well as multiple music-related servers. These servers included ones maintained by electronic music record labels such as YUKU, VISION, SATURATED, and VALE, as well as one from music producer and YouTube tutorial content creator Alckemy, and VST developer SixthSample. 51 responses were gathered over the course of two weeks, before the results were collected and examined.

Below, visualisations of the response data provided by Google Forms have been included. The quantitative data is shown with pie charts for the most part, while more qualitative data was compiled into the top common themes in each case. The process of compiling the qualitative data into common themes was greatly sped up by using OpenAI's ChatGPT. For example, for a question asking respondents what their favourite synthesizer was, the prompt given to ChatGPT was this:

"Below I have included a list of answers from a survey I conducted, asking what users' favourite synthesizer was. Each respondent is separated by a blank line. Could you list the three most common synthesizers including the count? Bear in mind, some answers include multiple synths, which you can include in the count."

This process was repeated for each qualitative set of answers, to quickly find the most common themes amongst the answers given.

Do you have much experience with synthesizers?

51 responses

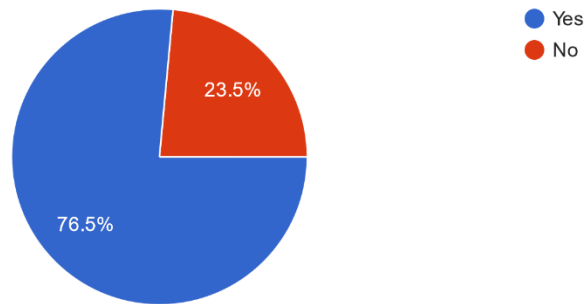


Figure 13. Results of the first survey question.

Since the survey was posted in multiple music-related Discord servers, most of the respondents had experience with synthesizers. This question was a branching one, changing the questions that followed. The following results will be from the respondents who answered yes, after which the results of the respondents who answered no will be gone through.

3.1.2.1 Questions for participants with synthesizer experience

Approximately how many synthesizers have you used?

39 responses

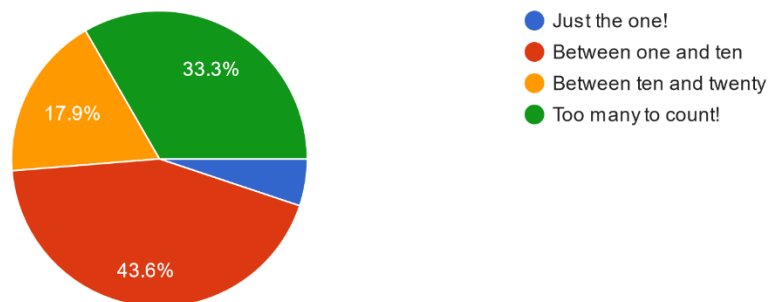


Figure 14. Results of the second survey question (yes branch).

Most of question two's respondents had used between one and ten synthesizers, but many had used too many to count. This question was asked to better judge the experience of each respondent.

Which do you use more often?

39 responses

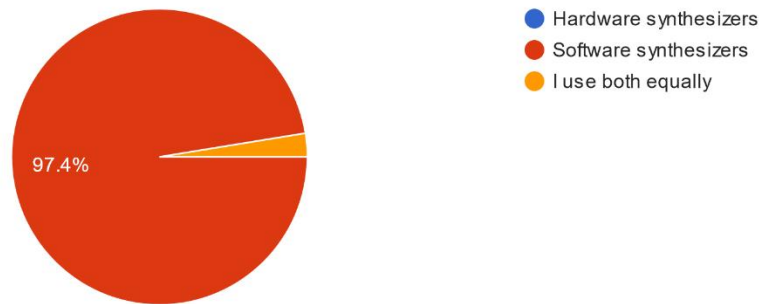


Figure 15. Results of the third survey question (yes branch)

A large majority of respondents use software synthesizers more often than hardware synthesizers, with no respondents answering hardware synthesizers, and only one stating that they use both equally. This was good because it meant the following answers would be informed by experience with the same type of synthesizer as MS24 would be.

The fourth question on the yes branch was “Do you have a favourite synth? If so, what is it?”. This was asked to get a better idea of the most popular software synthesizers, so that inspiration could be taken from them. ChatGPT was used to find the most common answers to this question, resulting in the following:

1. Serum (mentioned 8 times)
2. Vital (mentioned 7 times)
3. Phase Plant (mentioned 5 times)

These results were great to see, as inspiration had already been taken from Serum and Phase Plant during the initial design phase of the project, and Phase Plant had been researched during the gathering of similar applications during the initial requirements gathering stage. If any software synthesizers were to be used for inspiration from here on out, it would be one of these three.

The fifth question on the yes branch was “If you answered the last question, what do you like most about it?”. This was intended to get respondents to list the features of synthesizers that were most important to them. After formatting the results, they were given ChatGPT, which was asked to find the top three common themes, the result of which was as follows:

1. Versatility / Flexibility (mentioned 11 times)
2. Sound quality (mentioned 8 times)
3. Easy to use / Intuitive interface (mentioned 4 times)

The results from question five showed that the versatility, flexibility, and sound quality were the most important aspects of synthesizers for most respondents. MS24 would need to provide each of these features if it was to stand out as a decent synthesizer.

The sixth question on the yes branch was a matrix of Likert Scale questions asking users to rate which features of synthesizers were most important to them. In hindsight, this wasn't the best format for these questions, as Google Forms doesn't output graphs for each question but rather one extremely large graph, shown below:

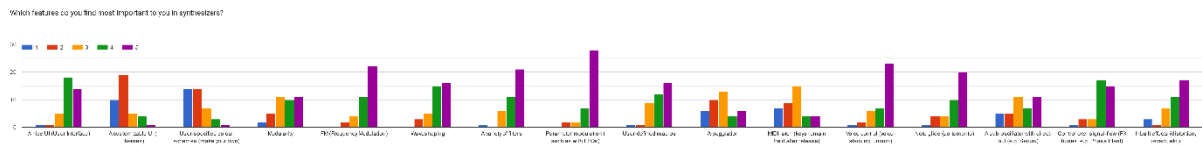


Figure 16. Results of the sixth survey question (yes branch)

To help find the most important features, ChatGPT was used again, this time being given each of the questions and sets of answers and asked to return the top features according to the respondents. The prompt used for this was a bit different to the others, to ensure accuracy:

“A Likert Scale matrix of questions was then asked in my survey. Below I have included each question in the matrix, each followed by a list of numbers representing the answers to those questions. Individual responses are separated by a line break.

You can disregard any empty lines between the numbers as these are from respondents who did not answer this question.

Also disregard "Which features do you find most important to you in synthesizers?" at the beginning of each question, the actual feature is contained in the square brackets immediately after.

The scores are as follows:

1. Not important whatsoever
2. Not very important
3. I don't care either way
4. Kind of important
5. Extremely important

Please return the features ranked from most important to least, providing the average score in each case.”

Here is the ranked list of features based on the average scores, returned by ChatGPT:

1. *Parameter modulation (such as with LFOs) - 4.86 average.*
2. *A variety of filters - 4.60 average.*
3. *User-defined macros - 4.35 average.*
4. *Wave shaping - 4.26 average.*
5. *FM (Frequency Modulation) - 4.16 average.*
6. *Modularity - 3.97 average.*
7. *A nice UI (User Interface) - 4.15 average.*
8. *User-specified colour schemes (make your own) - 2.47 average.*
9. *Arpeggiation - 2.81 average.*
10. *MIDI-latch (keys remain held after release) - 2.71 average.*
11. *A customizable UI (Themes) - 1.95 average.*

The results of question six would be used to prioritise different features and aspects of the application. It wasn't certain yet if all these features would be possible the chosen technologies, but they would all be attempted, and as many as possible would be included in the final application.

Would a cluttered UI put you off using a synth, or would you still attempt to learn it?

39 responses

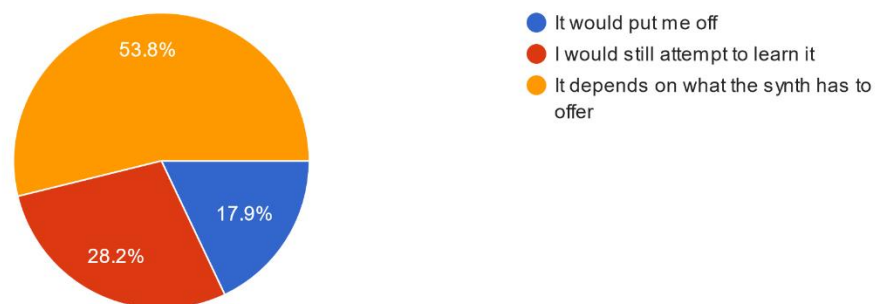


Figure 17. Results of the seventh survey question (yes branch)

When it came to whether a cluttered UI would put the respondent off using a synth, 53.8% of respondents stated that it depended on what a synthesizer had to offer, 28.2% stated they would

still attempt to learn it, and 17.9% stated that it would put them off. This was interesting, as it showed that the priority for most users is a synthesizer's features, not the user interface. If a synthesizer is good enough, users will put up with a cluttered user interface.

Which of the following are your preferred type of synthesizer?

39 responses

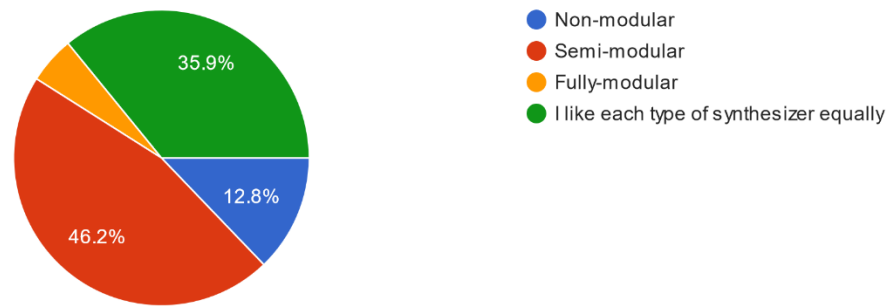


Figure 18. Results of the eighth survey question (yes branch)

Most of question eight's respondents stated that they preferred semi-modular synthesizers, which provide equal parts customisability and limitation. The next most common answer was "I like each type of synthesizer equally", which is probably because each type of synthesizer provides a unique experience, all of which are equally valid depending on the situation or mood the musician is in.

The ninth question was a follow-up to the eighth question, asking the user why they made their choice. The highly qualitative results of this question were fed into ChatGPT to find the most common themes. Below are the results:

- 1. Workflow and ease of use: Respondents emphasized the importance of a fast and simple workflow, as well as ease of use, especially for beginners. Some prefer non-modular synths because they are simple and straightforward, while others prefer semi-modulars for their versatility but not being too complex.*
- 2. Sound design and creativity: Many respondents preferred modular synths for their flexibility and experimentation, allowing for endless possibilities and happy accidents in sound design. Others prefer non-modular synths for their quick results, while some emphasized that the most important thing is the sound produced, regardless of the type of synth.*
- 3. Flexibility and control: Flexibility and control were also mentioned as important factors in choosing a synthesizer, but some respondents felt that the complexity of fully modular synths can be overwhelming and prefer a balanced approach.*
- 4. Learning and exploration: Some respondents expressed an interest in learning more about modular synthesis, while others want to explore and discover new sounds.*

Overall, the responses suggest that the choice of synthesizer type depends on personal preferences, intended use, and familiarity with the technology. The different types of synthesizers have their own strengths and limitations, and the most important factor is the sound produced and ease of use for the individual user.

As ChatGPT states, the responses to question nine suggest that the reason users use synthesizers with different degrees of modularity is mostly down to personal preference, intended use, and familiarity with the type of synthesizer. Most respondents noted that each type of synth has its own strengths and weaknesses, and the most important factor is not the modularity, but the sound produced and ease of use.

Would you agree that semi-modular synthesizers often strike a good balance between restriction and freedom?

39 responses

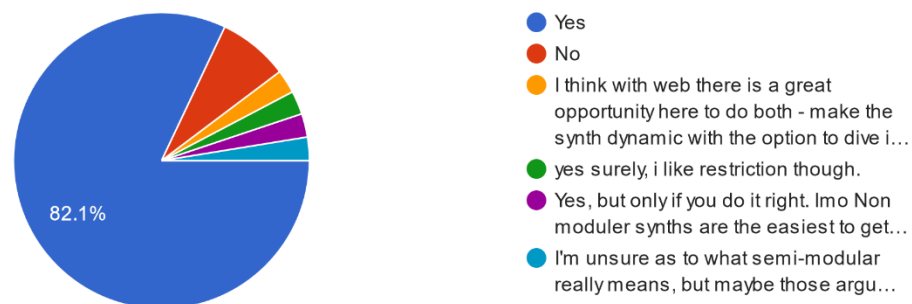


Figure 19. Results of the tenth survey question (yes branch)

The results of the tenth question showed that most users agree that semi-modular synths often strike a good balance between restriction and freedom. A small percentage didn't agree with this, and a few respondents used the "Other" option, which allowed them to add their thoughts. This question was mainly asked to know for sure if semi-modular was a good choice for MS24, and from the results given, it seemed that it was.

The eleventh question asks, "Are there any features you wish more synthesizers had?", which resulted in some lengthy answers. Again, ChatGPT was used to find the most common themes amongst the answers.

1. *Effects: Many respondents want more and better-quality effects, including built-in vocoders, granular synthesis, spectral warping, and convolution reverb. Others wish for an FX section that "actually" sounds good, with feedback and mix circuits in each effect, modulation of all parameters, and unusual filter types.*

2. *Modulation: Several respondents mention various types of modulation they would like to see, including key/MIDI-based modulation, audio rate modulation, using auto rate oscillators as modulation sources, random parameter modulation, and per-voice panning.*
3. *Randomization: Some respondents mention wanting more randomization features, such as smart randomization for creating patches, fully randomizing parameters over time, and loading random presets from a library.*
4. *Visualization: A few respondents mention wanting clearer visualizations of various synthesis techniques, such as FM effects, timbre, harmony visuals, and spectral morph/crossfade.*
5. *Other features: Other features mentioned include built-in limiter and clipper options on the main output, cross modulation, drawable curves envelopes for modulation, tuneable filters, MPE support/MIDI automation control, and a preset sharing website where users can browse sounds and see how patches are made.*

The results of question eleven show the most sought-after features in synthesizers according to the respondents. The first of these are effects, with lots of uncommon and somewhat exotic effects being listed, such as granular synthesis, spectral warping, and convolutional reverb. The next was modulation, with lots of respondents saying they would like to see more interesting methods of modulating parameters, such as key-based modulation and random parameter modulation (as opposed to cyclical LFO-based modulation). The third most popular category of features was randomization, with some calls for smart preset randomization. Visualisation was the fourth most popular category, with some respondents wanting clearer visualisations of synthesis techniques such as frequency modulation and harmonics. These results will be used as a source of inspiration for new features throughout the development of MS24.

Would a browser-based synthesizer interest you?

39 responses

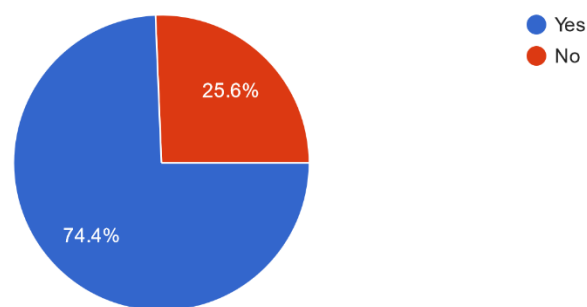


Figure 20. Results of the twelfth survey question (yes branch)

The twelfth question asked respondents if a web-based synthesizer would interest them. Approximately 75% of the responses were yes, which was great to see, but in hindsight it's not clear

what the intention behind this question was, other than to maybe prime the respondents' minds for the following question which asks questions specific to the planned web-based synthesizer.

The thirteenth question was another Likert Scale matrix of questions, this time asking the respondents "How important would the following features be to you in a browser-based synthesizer?". The results of this, like before, were visualised in the Google Forms results, but as one wide block of bar charts:

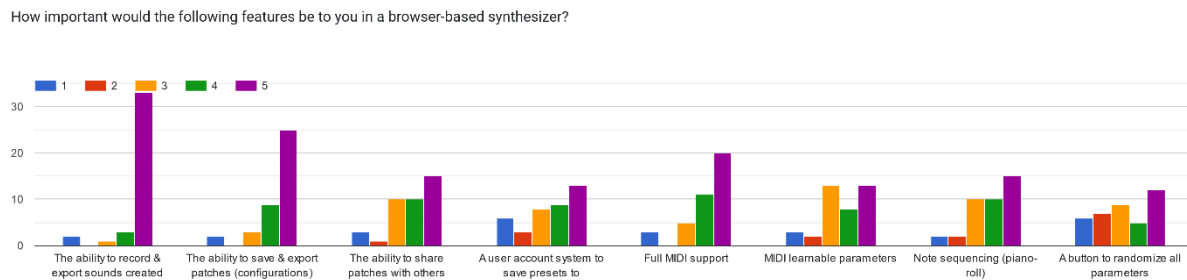


Figure 21. Results of the thirteenth survey question (yes branch)

ChatGPT was again given the results of each of these Likert Scale questions and asked to return the answers ranked from most important to least important, alongside the average score. The results of this were:

1. *The ability to record & export sounds created - 4.77 average.*
2. *The ability to save & export patches (configurations) - 4.29 average.*
3. *Full MIDI support - 4.18 average.*
4. *Note sequencing (piano-roll) - 4.07 average.*
5. *MIDI learnable parameters - 3.68 average.*
6. *The ability to share patches with others - 3.58 average.*
7. *A user account system to save presets to - 3.55 average.*
8. *A button to randomize all parameters - 3.34 average.*

These results made it clear which features planned for MS24 were of most importance to the respondents. It was obvious the application would need to be able to record and export any sounds created with it, so that they could be imported into a DAW for use in a music track or sound design session. The ability to save and export patches would also be very important, so that configurations made by users could be easily recovered between sessions. Full MIDI support was another high-ranking feature, meaning both MIDI-learnable controls and MIDI-key support would be required. Note sequencing was the last of the features which received an average score of over 4, however

this feature would no doubt be a difficult one to implement in a web interface, lots of research would have to be done if this were to be implemented.

The fourteenth question states the aim of MS24, and lists the planned features at the time:

- Two main oscillators
- FM between main oscillators
- One sub-oscillator
- A multi-modal filter
- An LFO for simple modulation
- A variety of effects
- An arpeggiator
- Glide controls

It then goes on to ask, “Is there anything else you'd like to see included or do you have suggestions for what's listed above?”. After giving the results of this to ChatGPT, the most common themes were listed:

1. Additional oscillator options: Several respondents suggested adding more than two oscillators, as well as a noise oscillator or a sub-oscillator.

2. Wavetable editor and granular synthesis: Respondents suggested adding a wavetable oscillator that can import user wavetables, as well as granular synthesis capabilities.

3. Envelope and LFO editing: Many respondents requested the ability to edit LFOs and envelopes, with some suggesting separate envelopes for the filter and amplitude.

4. Effects and filter options: Some respondents suggested adding filter drive (distortion) and more routable parameters, as well as a variety of effects.

5. User preset sharing and MIDI support: Respondents requested the ability to save and share patches and have MIDI support in the browser.

6. Additional modulation options: Several respondents requested additional modulation options, including ring mod distortion, OSC modulation, and more LFOs or modulation curves.

7. Other suggestions: Some respondents suggested adding a sampler, output clipper and limiter, a tuned filter, and a piano roll for melody writing.

The results of question fourteen would serve as another list of features to be considered for MS24. While some of these could be incredibly difficult to implement, it was useful to see what respondents felt were missing from the planned features at the time.

3.1.2.2 Questions for participants with no synthesizer experience

Are you interested in music production or sound design?

12 responses

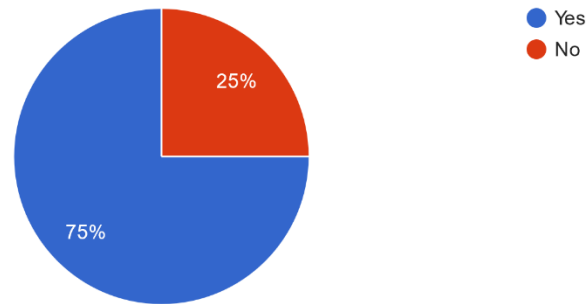


Figure 22. Results of the second survey question (no branch)

The first question asked of participants without synthesizer experience was intended to find the percentage of respondents who had an interest in the area, despite not having experience. Exactly three quarters of the respondents said they were.

Are you interested in music in general?

12 responses

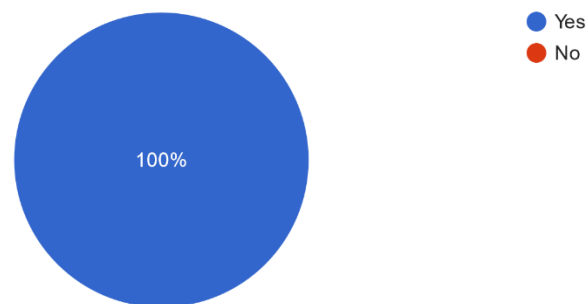


Figure 23. Results of the third survey question (no branch)

All twelve respondents with no synthesizer experience said they were interested in music in general.

Would you be interested in trying out a synthesizer?

12 responses

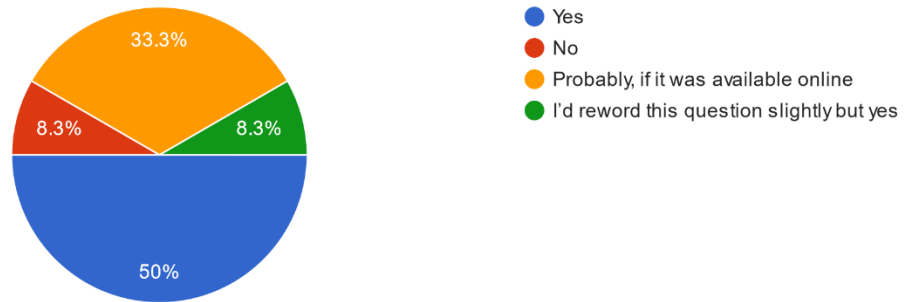


Figure 24. Results of the fourth survey question (no branch)

Seven of question four's respondents said they would be interested in trying out a synthesizer, while four said they would probably be interested if it was available online. Only one respondent said they wouldn't be interested.

For the fifth question asked on the no branch, participants were first shown three synthesizer interfaces before being asked to choose the interface they found most appealing.

The synthesizers chosen for this question were OB-Xd, a non-modular synthesizer based on the Oberheim OB-X hardware synthesizer, Diva, a semi-modular synthesizer created by U-He with a traditional looking interface, and Bazille, a modular synthesizer created by U-He which emulates modular hardware synthesizers by using virtual wires to connect components.

These interfaces were chosen as they show quite a broad range of interface complexity. Each of the interfaces shown to the participants have been included below.



Figure 25. OB-Xd's interface.



Figure 26. Diva's interface.



Figure 27. Bazille's interface.

Which one did you find most appealing?

12 responses

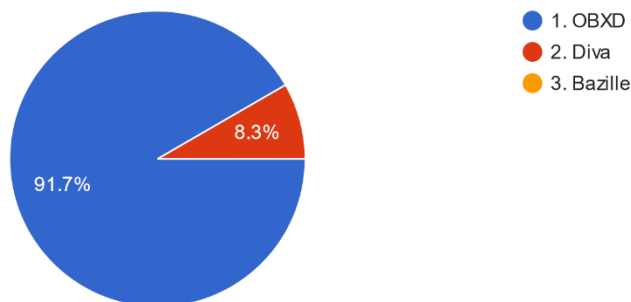


Figure 28. Results of the sixth survey question (no branch)

Unsurprisingly, most respondents found OB-Xd the most appealing, with only one respondent answering Diva. Understandably, nobody chose Bazille as the most appealing, as this interface would likely intimidate even experienced synthesizer users.

The sixth question on the no branch was a follow-up to the fifth question asking respondents why they made their choice. The common themes found by ChatGPT were as follows:

1. *Simple / easy to use interface (8)*
2. *Clearly labelled controls (3)*
3. *Fewer buttons/options (2)*
4. *Least cluttered interface (1)*
5. *Clear purpose (1)*

These responses made it clear that for MS24 to be appealing to users lacking synthesizer experience, its interface would need to be simple, providing clearly labelled controls, and not too many options.

Which one did you find least appealing?

12 responses

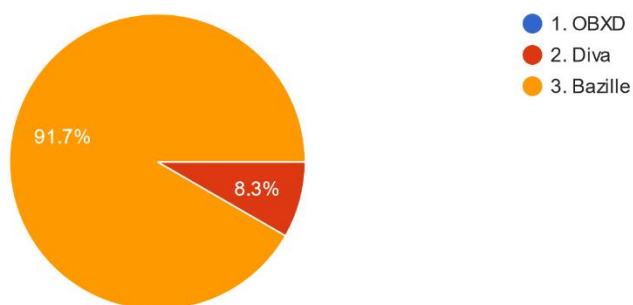


Figure 29. Results of the seventh survey question (no branch)

The participants were then asked to pick the interface they found least appealing, before being asked why it wasn't appealing to them in question eight. Eleven of the twelve respondents marked Bazille as being the least appealing interface, while one marked Diva as the least appealing.

The eighth question on the no branch was a follow-up to the seventh question asking respondents why they made their choice. The common themes found by ChatGPT were:

1. *Too much going on / Overwhelming (4)*
2. *Cluttered interface (2)*
3. *Complicated looking (1)*
4. *Unappealing colours (1)*
5. *Hard to differentiate sections (1)*
6. *Big / Lots of buttons / Intimidating (1)*

These responses further clarified the importance of a simple, uncluttered UI, if MS24 were to appeal to beginner synthesizer users.

Would a cluttered / intense looking UI (User Interface) put you off using a synth, or would you still attempt to learn it?

12 responses

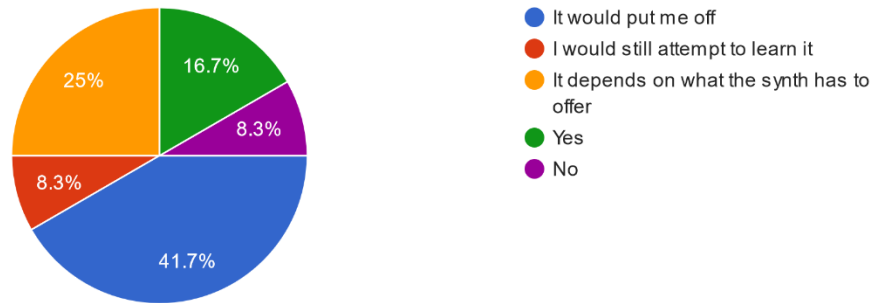


Figure 30. Results of the ninth survey question (no branch)

The ninth question asked on the no branch asked if a cluttered UI would put them off using a synthesizer. This question was rephrased shortly after the survey was released, which is why there are five answers shown in Figure 30. When combined, seven users said a cluttered UI would put them off, two said they would still attempt to learn it, and three said it would depend on what the synth had to offer. This suggested that inexperienced users were more sensitive to cluttered or intense looking UIs than experienced synthesizer users, as only 20% of experienced users said it would put them off.

Would a browser-based synthesizer interest you?

12 responses

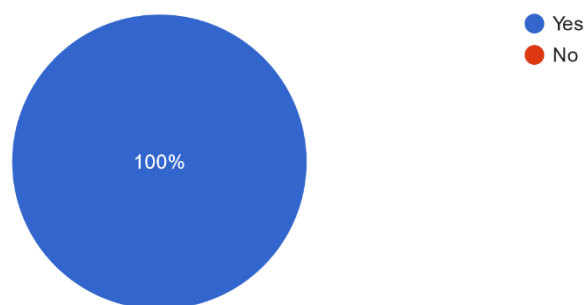


Figure 31. Results of the tenth survey question (no branch)

All twelve respondents on the no branch said they would be interested in a browser-based synthesizer. As with the yes branch, this question didn't have a clear intention, except for perhaps priming the respondents' minds for the next question.

How important would the following features be to you in a browser-based synthesizer?

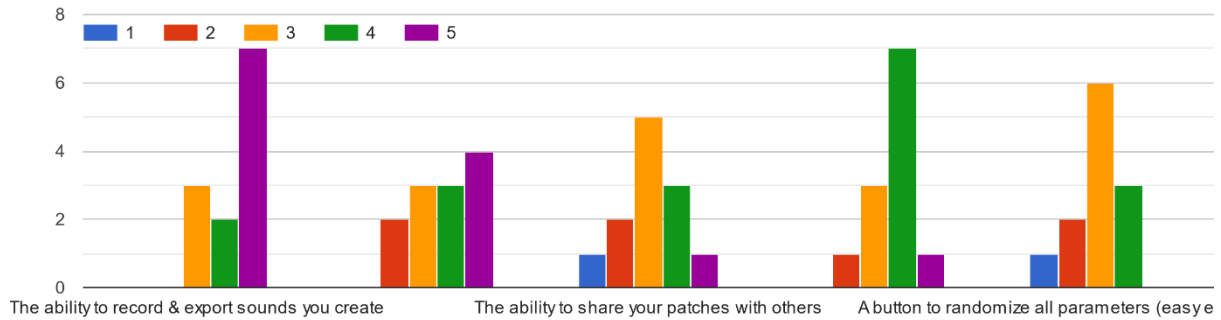


Figure 32. Results of the eleventh survey question (no branch)

Like the previous Likert Scale matrices, the results of this question were given to ChatGPT so that it could return the most common themes amongst the answers. The results were as follows:

1. *The ability to record & export sounds you create - 4.42 average.*
2. *The ability to save your synth patches (configurations) - 3.92 average.*
3. *A user account system to save patches to - 3.64 average.*
4. *The ability to share your patches with others - 3.17 average.*
5. *A button to randomize all parameters (easy experimentation!) - 2.92 average.*

These results were surprisingly similar to the results from the thirteenth question asked on the yes branch, although there were less options given to the participants on the no branch, as they were features that would have required a thorough explanation for people lacking synthesizer experience. The only difference in ranking were 3 and 4, which were ranked in reverse order by the experienced synthesizer users. Again, this clarified the importance of providing the ability to record sounds created with MS24, as well as saving synthesizer configurations for later use.

3.1.2.3 Questions for all participants

The last section of the survey was given to all participants, starting off by saying “Almost there! Please take a look at these MS24 hi-fi prototypes before answering the final questions”.

The prototypes which were provided have been included below.

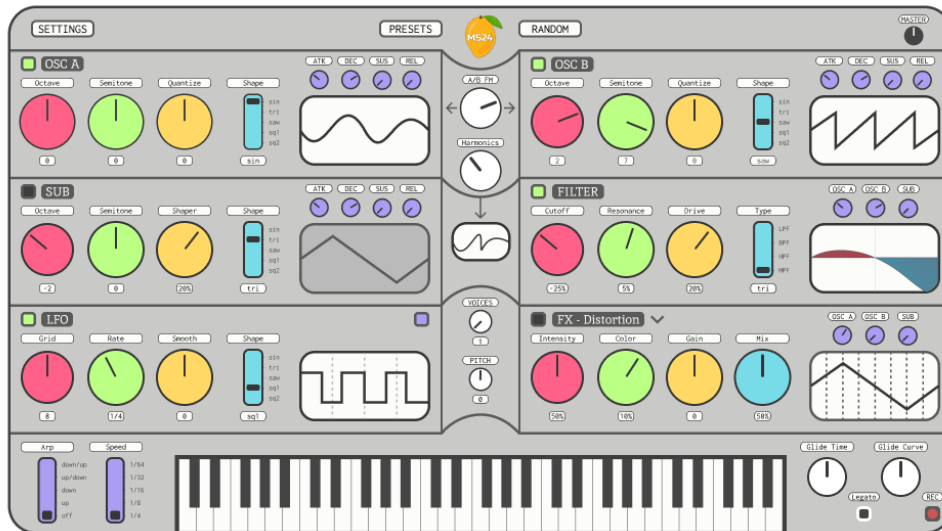


Figure 33. Version one of the proposed light themes.

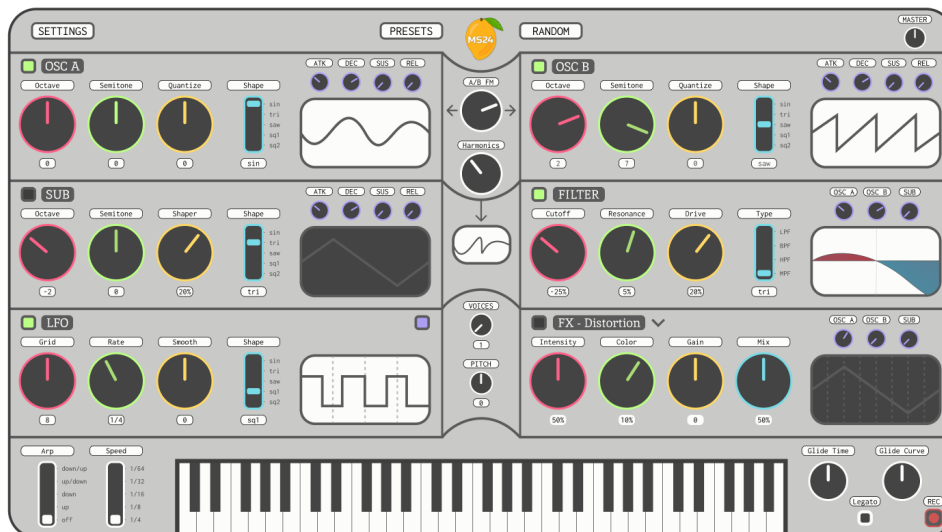


Figure 34. Version two of the proposed light themes.

These proposed designs were shown so that some feedback on the use of colours could be gathered, which would be used to inform decisions going forward. While this section could have been on its own unique branch, so that both participants with and without synthesizer experience could have been counted in the same results, sadly this was not realised until the survey had been launched, so the answers to the following question was split between each group of respondents. However, the

qualitative results from the final four questions were combined in Excel before being given to ChatGPT, resulting in the same difference.

Between the two versions of the light mode, which do you prefer?

39 responses

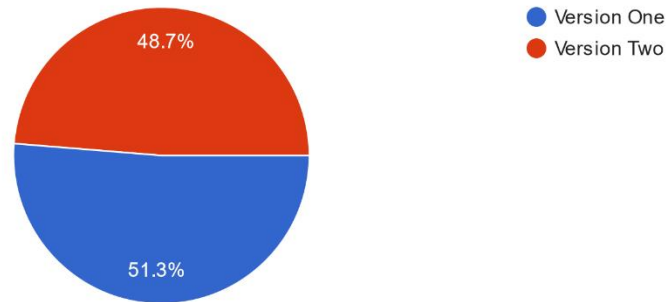


Figure 35. Results of the fifteenth survey question (yes branch)

Between the two versions of the light mode, which do you prefer?

12 responses

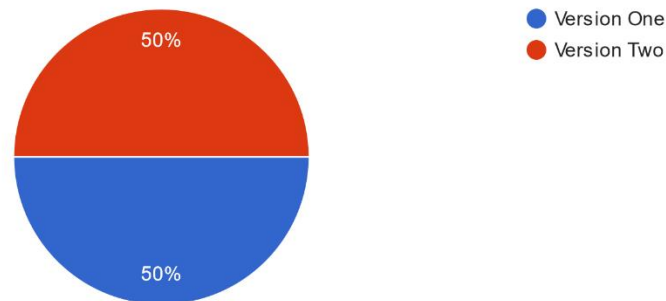


Figure 36. Results of the twelfth survey question (no branch)

Surprisingly, there was a 50/50 split between the two designs for both branches of the survey. This suggests that perhaps both colour schemes should be available to choose from in the final version of the application.

The question following this was “If there’s a specific reason for your answer, what is it?”.

When giving ChatGPT the results of this question, the answers to the previous questions were combined and provided alongside each of the answers to this one, so that the most common themes per-version could be found. The results have been included below.

Most common themes for Version One:		Most common themes for Version Two:
1. <i>Better Contrast (7)</i>		1. <i>Aesthetic (5)</i>
2. <i>Easier on the eyes (3)</i>		2. <i>Easier on the eyes (4)</i>
3. <i>Pleasant / Colourful (3)</i>		3. <i>Less busy / Less overwhelming colours (2)</i>
4. <i>More Obvious / Easier to distinguish (2)</i>		4. <i>Knobs look better and “more pro” (2)</i>
5. <i>Cleaner looking (2)</i>		5. <i>Cleaner looking (2)</i>

These results made it clear that each colour scheme had its own strengths and weaknesses, as well as the fact that it was largely down to personal preference, since three respondents who preferred version one and four respondents who preferred version two said they preferred that version as it was “easier on the eyes”. To appeal to all users, both styles would have to be incorporated into the application, giving users the ability to choose whichever one felt right to them.

Following the questions for the proposed designs, three optional questions were given on each branch of the survey, asking users if they had any other feedback on the design. The results from each branch were combined and then given to ChatGPT, which returned the most common themes from each set of results.

Top five response themes for “Things you like”:

1. *Simple design and layout (11)*
2. *Visuals and colours (7)*
3. *Approachable and easy to use (2)*
4. *Minimalistic design (2)*
5. *Dark mode (1)*

The most common things respondents liked about the proposed interface was the simple design and the colours. There would need to be a focus on keeping the application’s UI minimal and friendly.

Top five response themes for “Things you dislike”:

1. *Nothing disliked (5)*
2. *Dark mode colour scheme (3)*
3. *Dark mode keyboard colours (2)*
4. *Barren or generic appearance (2)*
5. *Text size(1)*

The most common things respondents disliked about the proposed interface was the dark mode’s colour scheme and specifically the dark mode’s inverted keyboard keys.

Top five response themes for “Things you would change”:

1. *Nothing (4)*
2. *Font size and readability (4)*
3. *Add more options/features (4)*
4. *Colours and contrast(3)*
5. *Clarifying the signal flow and reducing confusion (2)*

The most common things respondents said they would change were the font size and readability across the interface, and the colours and contrast. Four respondents did mention that they would have added more options or features, but this wasn't to do with the UI as asked. Readability and contrast would need to be prioritised.

Finally, each branch was given one last question, asking for general feedback.

Top response themes for “Lastly, do you have any general feedback?”:

1. *Positive feedback/encouragement (13)*
2. *No feedback / all good (4)*
3. *Suggestions for improvement (3)*
4. *Interest/excitement (3)*
5. *Request for open sourcing the project (1)*
6. *Request for keeping them updated (1)*
7. *Request for launching in DAWs as a VST (1)*

The general feedback given by users was mostly made up of highly encouraging messages, with a few even leaving their contact details so that they could be kept up to date with the progress of this project. A few suggestions for improvements were made too, but these didn't include anything that hadn't already been mentioned. One respondent requested the project be released as open source as they were working on a similar project in their spare time, while another suggested the synth get launched as a VST so it could be used in DAWs.

Overall, the results of the survey were extremely useful in gathering requirements for the application, and the feedback received from sending the survey to multiple Discord servers was very encouraging, sparking several conversations with software synthesizer and effect developers on Discord, two of which offered help should it be needed along the way, despite them not having any experience with web-based music technologies. The results of this survey would be used to inform future design decisions, and as a source of inspiration for any new features that might be added along the way.

3.2 Requirements modelling

The next step in the requirements phase is the requirements modelling phase. This phase begins with the creation of personas, which are fictional characters representing the different types of users in the expected userbase. This helps with the decisions around functional and non-functional requirements, which are the next steps in this phase. Functional requirements are things the application must be able to do to function properly, as well as to meet the needs of the userbase. Non-functional requirements, on the other hand, focus more on the usability and performance of an application, so that the best possible experience of using it can be attained.

3.2.1 Personas

Personas are fictional characters created based off the results of the survey to help with understanding the users' needs. They help to identify the ideal or relevant users of an application. For MS24, two main categories of users exist: those with plenty of experience using synthesizers, and those without. For the application to be successful, it will need to tend to the needs of both categories; simple and friendly enough to not put users with no synthesizer experience off, while versatile and powerful enough to not put users with lots of synthesizer experience off.

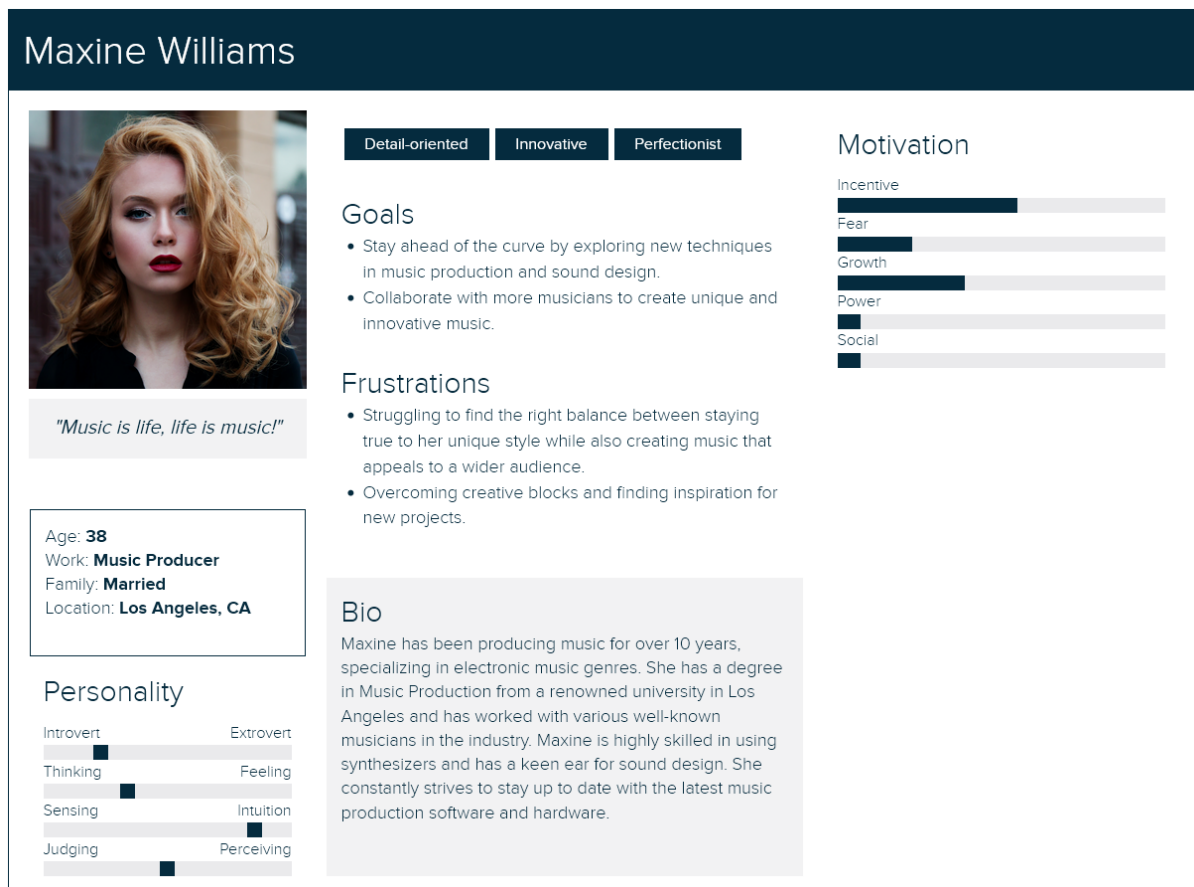



Figure 37. Professional Music Producer Persona, "Maxine Williams".

The professional music producer persona wants to stay ahead of the curve in the music scene by constantly exploring new ways to create her music, through experimentation with new techniques and technologies, and by collaborating with other musicians. MS24 can tend to these needs by providing a unique sound, with many ways to modulate and manipulate sounds made available. It could also help due to its remotely accessible nature, which traditional synthesizer software doesn't provide. This would allow for easier collaboration with other musicians.

David Rodriguez



"Let's learn and create something amazing."

Age: **22**
 Work: **College Student**
 Family: **Single**
 Location: **Austin, TX**

Creative
Curious
Open-minded

Goals

- Gain a basic understanding of music production and synthesizers.
- Learn how to create his own music and share it with others.

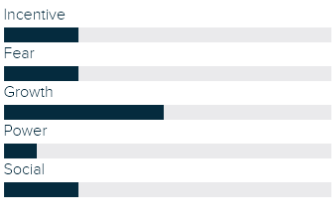
Frustrations

- Lack of knowledge and experience in music production and synthesizers.
- Finding the time and resources to learn and practice.

Bio

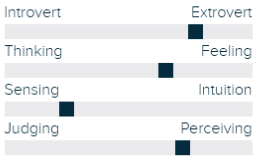
David is a college student studying marketing, with a passion for music. He has always enjoyed listening to music and attending concerts, but has never had any experience with music production or synthesizers. David wants to learn more about music production and hopes to eventually create his own music.

Motivation



Incentive	██████████
Fear	██████████
Growth	██████████
Power	██████████
Social	██████████

Personality



Introvert	██████████	Extrovert
Thinking	██████████	Feeling
Sensing	██████████	Intuition
Judging	██████████	Perceiving

The college student persona is studying marketing but has a great passion for music. He has always had a curiosity about music production but doesn't know much about it. The monetary investment required has put him off trying to learn, and as a result he has never used a synthesizer. MS24 could tend to his goals of wanting to gain a basic understanding of music production and synthesizers due to its free and accessible nature, not requiring any specific software or hardware to start experimenting, just an internet connection and a link to the application.

Another way MS24 could help is by providing tips for each component of the synthesizer, so that his learning process wouldn't be limited to just trial and error. Perhaps through using the application, David might discover that he really enjoys creating his own synthesizer configurations to generate unique sounds, or maybe he would realise that its not for him. Either way, MS24 would allow David to make an educated decision around the next step toward his passion for music.

3.2.2 Functional requirements

The following tables show the functional requirements which were gathered through the activities discussed above. The first of these tables show the “core” abilities of the synthesizer, ones that are fundamental to a functional subtractive FM synthesizer. The second table shows requirements that could be seen as extras; not exactly required for the synthesizer to function but having them would help the synthesizer stand out as being more versatile and feature rich.

Table 1. Functional requirements (high priority)

ID	Definition
Core FR 1.0	Takes PC keyboard input
Core FR 1.1	Takes mouse input
Core FR 1.2	Takes MIDI keyboard input
Core FR 2.0	Generates tones using oscillators
Core FR 2.1	Oscillator wave-shaping
Core FR 2.2	Oscillator frequency modulation (FM)
Core FR 2.3	Oscillators routed through ADSR amplitude envelopes
Core FR 2.4	Filters available to route audio through
Core FR 2.5	Effects available to route audio through

Table 2. Functional requirements (lower priority)

ID	Definition
Extra FR 1.0	Modulation of parameters with Low-Frequency Oscillator (LFO)
Extra FR 1.1	Arpeggiator available to create arpeggios from held keys
Extra FR 2.0	List of default presets available
Extra FR 2.1	User presets can be saved
Extra FR 2.1.1	User presets can be saved to user account (hosted)
Extra FR 2.2	User presets can be downloaded
Extra FR 3.0	Audio can be recorded and exported
Extra FR 4.0	Interface theming
Extra FR 5.0	Oscilloscope visualisations

3.2.3 Non-functional requirements

Non-functional requirements are requirements which if not met do not stop the application from working but mean that the application is not working as well as it should. The following tables show the non-functional requirements in terms of usability and performance.

Table 3. Non-functional requirements (usability)

ID	Definition
NFR 1.0	The synthesizer interface should be intuitive and easy to navigate for both novice and expert synthesizer users.
NFR 2.0	The synthesizer should have a semi-responsive design that works well on different screen sizes and resolutions. However, mobile friendliness may not be obtainable.

NFR 3.0	The user should be able to save their presets and settings for future use.
NFR 4.0	The synthesizer should be accessible to people with disabilities, such as support for keyboard-only navigation and screen readers.
NFR 5.0	The synthesizer should have clear and concise documentation for users and developers.

Table 4. Non-functional requirements (performance)

ID	Definition
NFR 6.0	The synthesizer should be fast and responsive, with minimal latency and delay in both the user interface and audio output.
NFR 7.0	The synthesizer should support multiple simultaneous voices and complex polyphony without audio glitches or crashes.
NFR 8.0	The synthesizer should use minimal system resources to avoid putting a strain on the user's device.
NFR 9.0	The synthesizer should have a low error rate and be able to recover from errors quickly.

3.3 Feasibility

When it comes to the feasibility of this project, creating a web-based synthesizer using React, Tailwind CSS, Vite, webaudio-controls, and Tone.js is technically feasible. Most of these technologies have been widely adopted and are well-established, with extensive documentation and support available for all but the webaudio-controls JavaScript library.

However, there will be some challenges to consider. One potential issue is browser compatibility. As with any web application, MS24 may not function the same way on all browsers. While React is relatively consistent across modern browsers, the Web Audio API under Tone.js may behave differently depending on the JavaScript engine the browser uses, which could lead to certain compatibility issues. Additionally, the webaudio-controls library may not function as expected on some browsers, as it is an old unmaintained library with a jQuery API. Therefore, testing the application thoroughly across multiple browsers and platforms throughout the development process will be very important.

Another potential issue is the performance of the synthesizer. Software synthesizers can be very computationally intensive, so running them in a web browser may pose performance issues due to high CPU load. Reducing load will be even more important than with regular web applications as audio is a delicate medium which makes performance issues easily noticeable due to stutters and glitches. For this reason, it will be essential to optimize the code, minimizing unnecessary computations to ensure the application runs smoothly on most devices.

Finally, the complexity of the application's logic may also be a challenge in terms of development time and effort, requiring a high degree of functionality and a strong focus on user interface design. Therefore, thorough examination of documentation, careful planning, and iterative development will be essential to ensure the application meets the requirements set out in this chapter.

In conclusion, creating a web-based synthesizer using React, Tailwind CSS, Vite, webaudio-controls, and Tone.js is certainly feasible, but there are a few challenges to consider. Careful planning and optimizing the code for performance will be necessary to ensure the project's success.

3.4 Conclusion

In conclusion, this chapter provided an overview of the process of gathering and modelling the requirements of the application. The chapter is divided into three main sections: Requirements Gathering, Requirements Modelling, and Feasibility.

In the Requirements Gathering section, the process of how requirements were gathered was discussed. First, similar applications were examined to get a better understanding of the common features and functionalities of software synthesizers and more specifically, web-based software synthesizers. A survey was then conducted on Google Forms which received a total of 51 responses. The results of this survey were then analysed so that the preferences of potential users could be obtained.

In the Requirements Modelling section, the process of modelling these requirements was discussed. Personas were created to represent the different types of users that might use the application, before the functional and non-functional requirements were created, informed by the requirements gathering stage.

Lastly, in the Feasibility section, the feasibility of the technologies used was examined. The technical feasibility was assessed, with possible challenges highlighted, as well as what should be considered to ensure the project's success.

4 Design

This chapter describes the design phase of the project. The purpose of the design phase is to allow for developers to arrive at a design for their application so that the application can be implemented in such a way that it meets the requirements set during the requirements gathering stage.

4.1 Program Design

Program design refers to the design required to make the task of programming and coding of the application more straightforward. This section is split into five parts: Technologies, Structure, Design Patterns, and Process Design.

Two versions of the Technologies and Structure section are included, as the technology stack was adjusted during Sprint 4 of the implementation phase. The first version was written during Sprint 2, while the second version was written during Sprint 8. The Design Patterns, Application Architecture, and Process Design sections were also adjusted to explain both versions at once, as the differences weren't great enough to justify entirely separate sections.

4.1.1 Technologies (V1)

The technologies chosen for MS24 are as follows:

- **React** – JavaScript framework
- **ToneJS** – A Web Audio API library
- **'webaudio-controls'** - GUI library
- **Vite** – Frontend build tool
- **Zustand** – State management library

React was chosen as it has been the JavaScript framework with the highest usage for more than five years, according to the annual State of JavaScript survey (*State of JavaScript, 2022*). Learning to use React earlier in the academic year demonstrated the power of its state management methods, which would come in handy if the web application was to provide user account functionality.

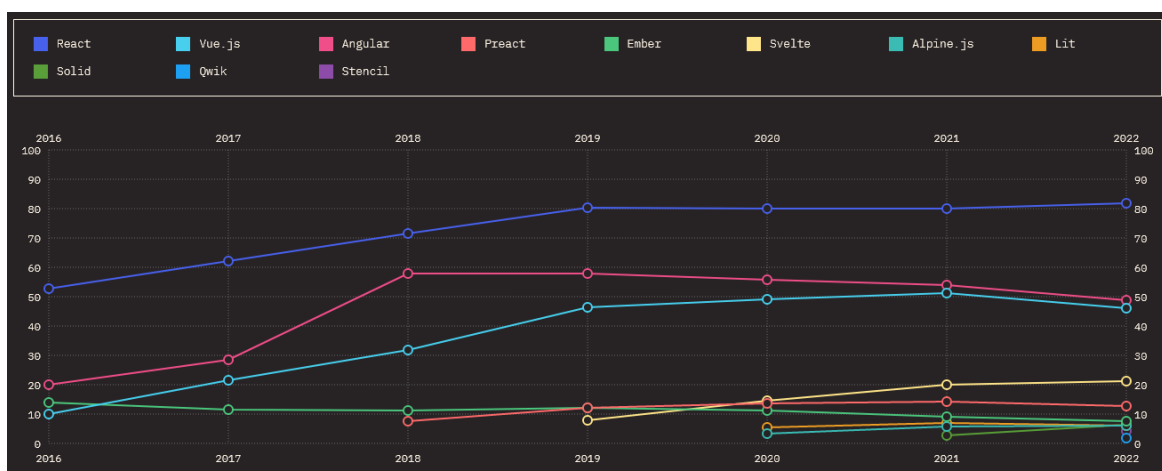


Figure 38. State of JavaScript 2022 framework usage graphs (React in blue).

Tone.js was chosen because it acts as an abstraction of the Web Audio API, providing many components found in Digital Audio Workspaces (DAWs) through its premade classes. The most important of these would be a transport system, which would prove to be very important when it came to keeping all audio events in sync with each other, a task which would otherwise be difficult within a browser. It also provides many nodal configurations, such as its Synth classes, which are made up of various Web Audio API node trees, such as oscillator nodes connected to amplitude envelope nodes. Using Tone would vastly speed up development time, since many advanced components were available from the get-go.

The 'webaudio-controls' JavaScript library was chosen as it was the best-looking GUI library that could be found which provided elements often found on synthesizer interfaces, such as knobs (also known as 'pots' or 'rotary encoders'), sliders, and a customisable GUI keyboard. Other libraries providing this were hard to find, and those that did, such as interface.js or p5.touchgui, were deemed inadequate.

Vite was chosen as the build tool for the application. This is mostly because Vite has garnered an incredible amount of popularity since its release in 2020, with it topping the State of JavaScript 2022 build tool retention charts and appearing on numerous YouTube channels over the past two years. This is because Vite aims to provide a versatile, quick, and efficient development experience by using native ES modules instead of traditional bundling methods like used by Webpack or Parcel. Boasting "Lightning Fast" Hot Module Replacement (HMR) and highly optimized application builds, it seemed like the best choice for an application with such a priority for performance.

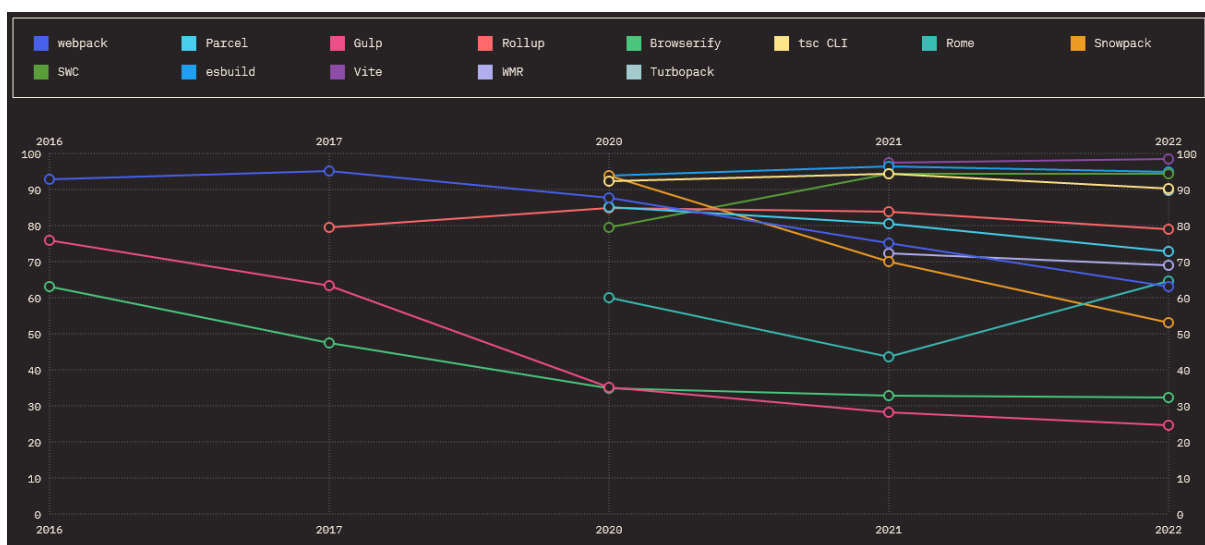


Figure 39. State of JavaScript 2022 build tool usage graphs (Vite in purple).

Zustand was chosen as the state management library for React as it is an incredibly lightweight library which greatly simplifies React state management, and provides an unopinionated API based on hooks, allowing for great customisability. This would help with creating a global store for all application state to be contained in, making it easily accessible across each of the files in the project, which would be very important when managing the state of so many controls and audio elements.

Other technologies which could have been used include the Web Audio API in place of Tone, various other JavaScript GUI libraries such as interface.js or p5.touchgui for interactive GUI elements, Vue or another JavaScript framework (or even vanilla JavaScript) for the frontend, and Redux or Jotai for state management.

Some of these technologies, such as the Web Audio API and Redux were not suitable because of the complexity involved in implementing them would be far greater than the benefits they would provide when developing such a thin web app.

With the Web Audio API, while it would be possible to do anything Tone.js can do, if not more, it would be incredibly time consuming to develop certain core elements of synthesizers without Tone, such as arpeggiation, which relies on the synthesizer being in sync with a clock. This clock is tied to a particular BPM (Beats-per-minute) which governs synchronicity across all audio signals.

In the case of Redux, it would have overcomplicated management of state, while unnecessarily increasing the overall package size. These technologies are more suited to large, complex projects that require more advanced state management.

A library called React Flow was also considered as the main GUI library when originally trying to decide between a semi-modular or fully modular synthesizer. This is because it is the framework behind the likes of Figma, providing a powerful drag-and-drop interface which would have been perfect for a fully modular design. However, after experimenting with the library for a while, it was deemed unsuitable for the project, due to the complexity of customising it for more niche use-cases like this one.

4.1.2 Structure (V1)

4.1.2.1 React

React is a JavaScript framework for building user interfaces. It uses a component-based architecture, meaning a React project typically consists of a set of large, unique components (which could be seen as pages) constructed from small, reusable components (such as custom input boxes).

Each component represents a piece of the UI and can manage its own state and render its own HTML. React uses JSX syntax to render the components, which allows for easy integration of HTML and JavaScript in a single file.

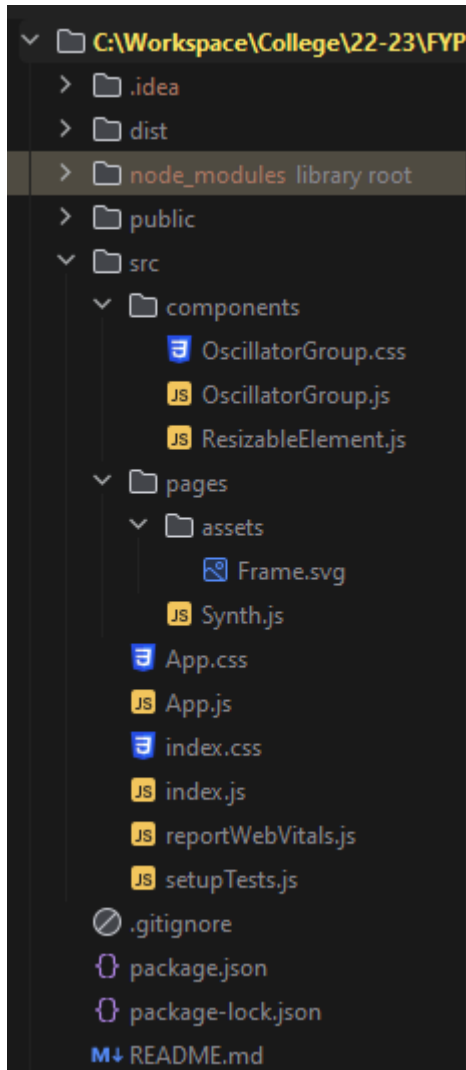
A React component takes inputs in the form of props and returns a description of the user interface in the form of a virtual DOM (Document Object Model). The virtual DOM is then used to render the user interface in the browser.

The state of the application, which is the data that changes over time, is managed using state variables. When the state changes, React automatically updates the user interface to reflect the new state.

React also provides a router, which is a mechanism for managing the navigation between different pages or sections of the application using routes. A route is a mapping between a URL path and a component. This is necessary because React creates single-page applications (SPAs) by using a combination of client-side rendering and a virtual DOM. A single-page application is an application that runs within a single web page and dynamically updates the page without reloading the entire

page. The URL might change when navigating between pages, but this is just instructing React what to automatically inject into the single `index.html` file being used behind the scenes.

While React’s documentation states that it “doesn’t have opinions on how you put files into folders”, an unwritten standard in the React ecosystem is that projects often use the following structure:



`public` contains static assets such as the `index.html` file (where code is dynamically rewritten) and favicons (used for the tab icon & icons on mobile).

`src` contains all the source code for the application, including components, routes, hooks, controllers, and views.

`src/components` contains all the reusable components used in the application.

`src/pages` defines the application's routes, which determine what major component is rendered based on the URL.

`src/controllers` (not included in screenshot) contains the logic for controlling the application's state, such as updating the state in response to user actions.

`App.js` contains the logic for the `App` component, which is the main component in React, acting as a container for all other components.

`index.js` contains logic corresponding to `index.html` inside the `public` folder. This handles the dynamic injection of code from `App.js`, providing the SPA functionality.

Figure 40. A screenshot of MS24's folder structure (React)

4.1.2.2 `Tone.js`

`Tone.js` is a JavaScript library for audio synthesis and processing. It provides a wide range of built-in components for creating musical compositions and sound design. The structure of `Tone.js` is centred around the concept of modules, which are the building blocks of audio processing and synthesis, made up of nodes provided by the Web Audio API.

Each module in `Tone.js` represents a specific audio processing or synthesis component, such as oscillators, filters, and effects. Modules can then be connected to form complex audio processing

chains. Tone.js provides a simple, intuitive API for creating and connecting modules, as well as for controlling their parameters in real-time.

The main components of Tone.js include:

- **Sources/Generators:** These are modules that generate audio signals, such as oscillators, synthesizers, and noise generators.
- **Effects:** These are modules that process audio signals, such as delays, reverb, and distortion.
- **Controllers:** These are modules that control the parameters of other modules, such as envelopes, LFOs, and sequencers.
- **Core:** This is the central component that manages the audio context and provides essential functionality for Tone.js, such as event scheduling and timing.

By combining these components with the methods provided, it will be possible to build MS24's internal audio architecture.

4.1.2.3 Zustand

Zustand is a minimalistic state management library for React. Designed to be fast and easy to use, the main structure of Zustand is based on the concept of a store, which is where the stateful data of the application is kept.

A Zustand store is a JavaScript object which holds the application's state. The store can be created using the **'create'** function provided by Zustand. The state in the store can be updated by dispatching actions, which are functions that modify this state.

The store can be used in React components by wrapping the components with the **'useStore'** hook. This hook provides access to the state in the store and allows components to subscribe to updates to state.

In a Zustand-based React application, the store is the "single source of truth" for the application's state, according to Zustand's documentation. This helps in keeping the state management simple and organized. Components can update the state by dispatching actions, and other components can subscribe to updates to the state and re-render when the state changes.

4.1.3 Technologies (V2)

The technologies chosen for the updated version of MS24 are as follows:

- **Vanilla JavaScript** – JavaScript framework
- **TailwindCSS** – CSS framework
- **ToneJS** – A Web Audio API library
- **'webaudio-controls'** - GUI library
- **Vite** – Frontend build tool

Towards the end of Sprint 3, it was realised that React had been overcomplicating the development process. While the framework was picked for its powerful componentization and state management, it turned out that Tone.js and webaudio-controls managed their own state in many ways, and Tone's Audio Context didn't like to be passed between components. With state management and componentization unusable, React only served as an obstacle in the process, with plenty of code already having been written which bypassed React's use cases, such as traditional event listeners which aren't very compatible with React's event handling and useEffect hooks.

Vanilla JavaScript was chosen instead, as this would vastly reduce the troubleshooting factors when problems were encountered, as up until this point it was unclear what was causing a lot of the issues when they were encountered, causing lengthy troubleshooting processes which were difficult to resolve.

TailwindCSS was the only other difference between version one and two of MS24's technology stack. This CSS framework provides a set of predefined CSS classes allowing HTML elements to be easily styled without writing custom CSS. This allows for much faster development while still providing a large amount of customisability. It is also safer from an interface design point of view, as any spacing and font sizing classes are consistent with a standardized set of styles following tried and tested design patterns. It is also very customisable, allowing for the addition of custom classes and themes, as well as overrides for defaults through its configuration file.

4.1.4 Structure (V2)

The primary difference in structure between versions one and two of MS24 is the migration from React to Vanilla JavaScript. Instead of componentizing any reusable parts of the application into separate files which contain both JavaScript and HTML code through JSX, the traditional approach of having one HTML file per page was taken, with a singular JavaScript file containing all the internal logic being imported between the head tags.

Since Vite was being used still, with a custom configuration retrieved from GitHub specifically for use with Vanilla JavaScript and TailwindCSS, there was still a `/public` folder, however this time its only use was for containing files which Vite should include in application builds, such as MS24 configurations and the webaudio-controls library, since only JavaScript modules can be bundled by Vite.

The only other files included would be configuration files for the various libraries in use, such as `package.json` containing instructions for Node Package Manager (NPM), so that the install command would download and install the right dependencies, `vite.config.cjs` for configuring Vite to work with Vanilla and Tailwind, `tailwind.config.cjs` for configuring Tailwind to work with Vite, and `postcss.config.cjs` for Tailwind's dependencies.

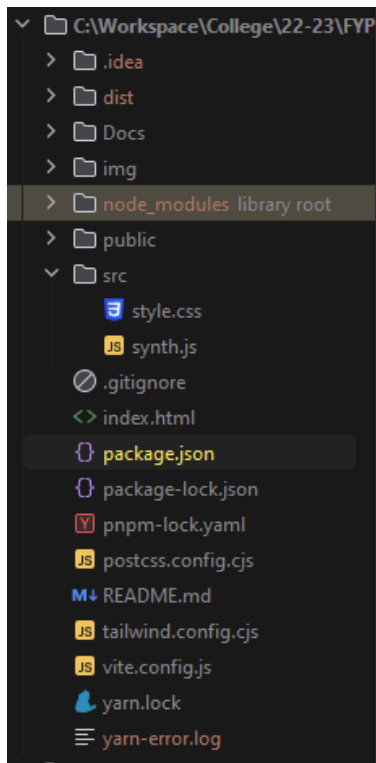


Figure 41. A screenshot of MS24's folder structure (Vanilla)

4.1.5 Programming Paradigms

JavaScript can be seen as a multi-paradigm language, as it allows programmers to mix and match Object-Oriented Programming (OOP), Functional Programming (FR), and Procedural Programming (PR) paradigms. The same goes for React, although with more of a focus on OOP and FR.

On one hand, React is built on OOP principles, specifically through the concept of encapsulation. React components are like objects, each one having its own state and logic, managed independently of other components. This helps keep the codebase organised, modular, and makes it easier to maintain.

On the other hand, React also uses functional programming principles, specifically through the concept of immutability. React encourages developers to write pure functions, which don't have side effects and always return the same output when given the same input. This makes it easier to ensure that the code is correct. React also relies heavily on functional programming concepts such as higher order functions, especially through its Higher-Order Components (or HOCs), which are components that can take other components as arguments, outputting entirely new ones.

4.1.6 Design Patterns

While React isn't a Model-View-Controller (MVC) framework, it can still be used to create applications incorporating an MVC-like design pattern.

In the MVC pattern, the Model represents the data and logic of the application, the View represents what data is displayed to the user, and the Controller mediates between the two. React can be used to implement this pattern by using components as the building blocks of the user interface.

Each component can be thought of as a combination of Model, View, and Controller, with the state representing the Model, the render method defining the View, and its event handlers acting as the Controller.

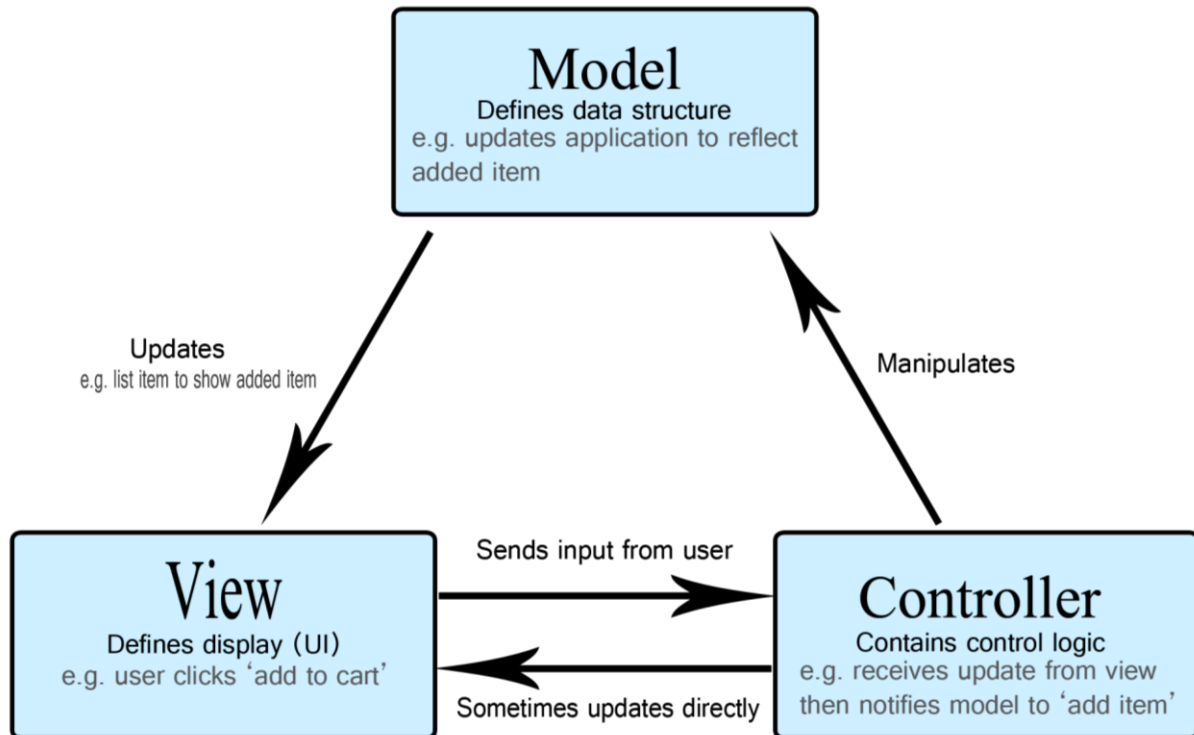


Figure 42. The Model-View-Controller design pattern (<https://developer.mozilla.org/en-US/docs/Glossary/MVC>)

The same applies to traditional web applications using Vanilla JavaScript, HTML and CSS.

In MS24's case, the state of the application, consisting of all the individual parameters across the synth can be thought of as the model. The graphical user interface and styling rendered in the DOM can be thought of as the view, and the Tone.js logic responsible for audio synthesis and modulation as well as the keyboard logic made up of event listeners can be thought of as the controller.

4.1.7 Application Architecture

The technologies initially selected for building MS24 were React, Vite, Tone.js, webaudio-controls, and Zustand. As explained earlier, from Sprint 4 onwards Vanilla JavaScript replaced React and Zustand. These technologies were chosen based on their ability to provide the necessary building blocks for the application, as well as due their popularity and community support.

This bundle of technologies can be seen as a "technology stack" or "software stack", a bundle of software and libraries that make up an application. There are a huge number of different stacks out there, but web development stacks will be the focus here since that is what MS24's stack is.

This isn't a traditional web development stack by any means, especially because it only accounts for a frontend. Popular web stacks such as MERN (Mongo, Express, React, Node) or MEVN (Mongo,

Express, Vue, Node) consist of a frontend (React or Vue), server (Express and Node), and database (Mongo), and as such are known as “full stacks”.

If MS24 were to provide user account functionality, so that users could register an account and save their synth configurations to that account so that they could be retrieved from anywhere once they logged in again, then these backend technologies would have made it a full-stack, but since MS24 is to be a powerful web-based synthesizer first and foremost, the stack is just a complex frontend for the time being.

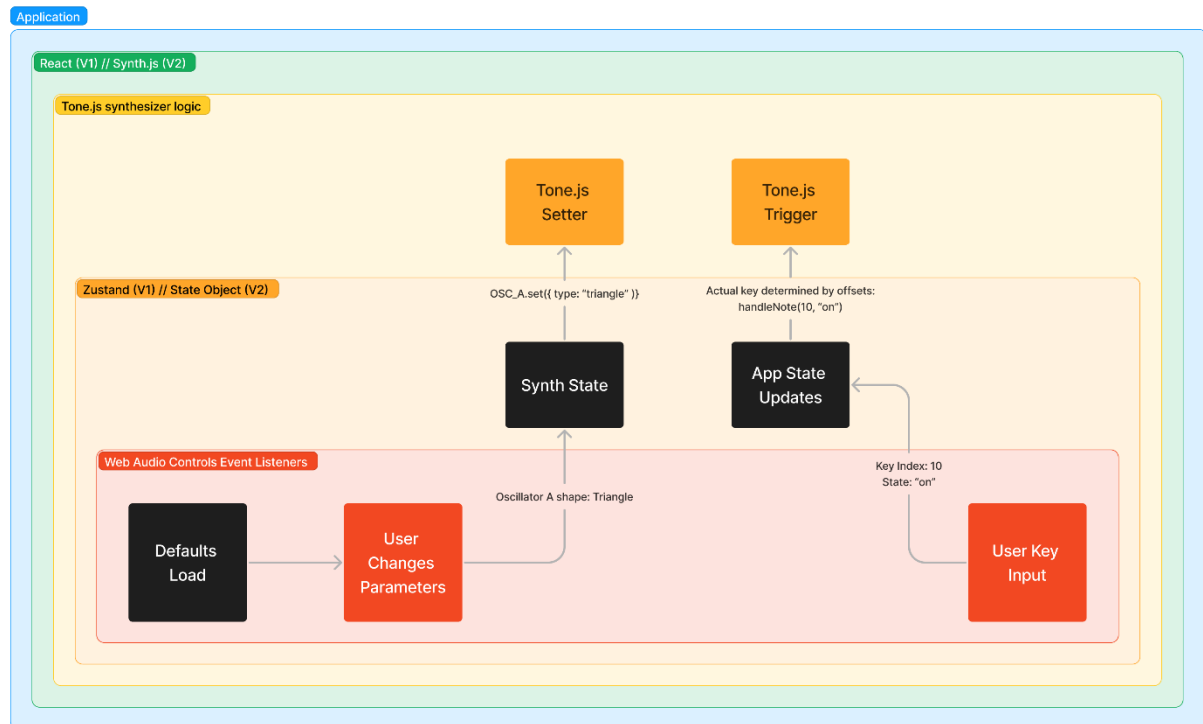


Figure 43. A block diagram of MS24

4.1.8 Process design

Process design refers to the planning of an application through the sequences of steps required for the logic to be developed. This involves identifying the inputs, outputs, and all logic in-between. For MS24, the basic flow of data will be planned in text format as well as through diagrams before pseudocode is written to give more detail to the intended logic.

4.1.8.1 Basic Flow (Text-format) (Pseudocode)

1. Initial User Input (Consent required for Web Audio to function)
2. Note Input (Either with mouse, keyboard, or MIDI controller)
3. Oscillators (Primary tone generators)
4. ADSR (Amplitude envelope of the oscillators)
5. Filter (Changes the characteristics of audio generated)
6. FX (Changes the characteristics of audio generated)

7. Output (Audio is played through default sound device (i.e., user's speakers))

4.1.8.2 Basic Flow (Flow-chart)

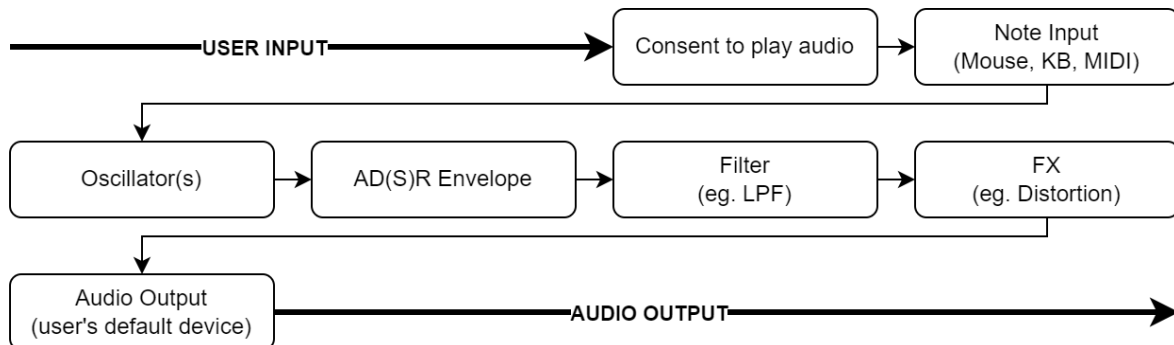


Figure 44. Flow chart showing the signal flow of the application.

4.1.8.3 Pseudocode (V1)

When the app first loads, the user will be presented with a welcome screen, including a button asking them to consent for the Web Audio API to play sounds. This is because browsers have started to implement an ‘autoplay policy’ which will eventually restrict Web Audio contexts from automatically playing audio. It isn’t fully implemented, currently only resulting in console warning, but to prevent the app breaking in the future, the audio context should always be started by the user. (MozDevNet, *Autoplay guide for media and web audio apis - web media technologies: MDN*)

This “page” is the main React App component, and it is only when the user starts the context that it renders the Synth component. This component contains all the stateful synthesizer JSON data, Tone logic, state management logic, and webaudio-controls custom HTML elements, through JSX.

As soon as the Synth component loads, two primary JSON objects are initialised. The first is *defaultAppData*, which contains information about the web app itself, such as the Booleans *domLoaded*, *windowLoaded*, and *toneRunning*. The second is *defaultSynthSettings*, which contains objects representing each part of the synthesizer, each containing the default Tone settings and the default webaudio-controls values. These JSON objects are then used to create Zustand stores, each with their own custom mutators to allow for this data to be updated on the fly.

Once the event listeners tied to the window and DOM load events trigger, the appData store Booleans are updated to true. This allows the webaudio-controls JSX to have their own event listeners attached to them. If they are assigned event listeners before the window has finished loading all elements, the app fails to find the elements in question and crashes.

The user is now presented with the main synthesizer GUI containing many knobs, sliders, and buttons to interact with, as well as a musical keyboard. If the user changes any of the parameters, this triggers a store update as well as a Tone ‘set’ function. This set function updates the corresponding Tone audio parameter in real time, which means the changes can be heard while the user is playing a note.

On the topic of playing notes, when the user plays a key on either the onscreen keyboard or their own physical keyboard, this sends an ‘attack’ signal to Tone containing the note information of that

key, which fires that note using the current synthesizer configuration. Once the user releases that key, a 'release' signal is sent to Tone, telling it to stop playing that note.

4.1.8.4 Pseudocode (V2)

Just like version one of MS24, when the app first loads, the user is presented with a welcome screen. This welcome screen includes instructions for using the synthesizer, such as keyboard shortcuts for the GUI controls and which keys emulate the notes on a musical keyboard. To progress to the main synthesizer GUI, the user must first click the button asking them to consent to the application playing audio.

Once the user clicks this button, an event listener is triggered which replaces the main container's content with the main synth GUI, made up of many div elements using flexbox for positioning, each containing several webaudio-controls knobs and sliders, as well as the virtual keyboard positioned at the bottom of the screen. Each of the controls' defaults, such as their default value, minimum and maximum values, and how much they increment by is loaded when the HTML loads, as these properties are set when initialising the elements. At the same time, the synth.js script loads its own defaults with several objects defined at the top of the script. These objects match the HTML defaults, mimicking the state of the synth, but not actually set using the element's properties. Tone.js objects are then initialized with the values from the objects representing the synth's default state. By this point, the GUI, the state objects, and Tone.js all match up.

If the user triggers the synth at this point, by clicking the GUI keyboard with their mouse, or by pressing keys on their computer keyboard or MIDI keyboard, these default settings are used by Tone in combination with the value of the key that was triggered, resulting in a basic sound playing through the users' default sound device.

Once the user uses their mouse to adjust any of the controls, an event listener is triggered for that control, updating the corresponding property in the state object. A function called connectTone is then triggered, which re-sets and reconnects Tone using the newly updated object properties.

If, while playing notes, the user changes a control responsible for determining the frequency generated by one of the oscillators, such as an octave or semitone offset, the state is adjusted like before, but a function called changeNote is also triggered, which tells Tone.js to stop playing the original frequency and then immediately trigger the new frequency. Without this function, the state would be updated before the note playing had been fully handled. This would mean that if the user let go of the key they were holding, the logic would tell Tone to stop playing the wrong key, as it would be referring to the updated state, while the internal state no longer reflects it.

This sort of stateful logic is one of the main challenges of working with the Web Audio API, as it is an uncommunicative API, meaning its own internal state isn't accessible. To ensure the application is bug-free, the application's state must always match the internal state, so that the correct logic can always be determined correctly.

4.2 User interface design

This section presents the user interface (UI) design for MS24, from initial paper prototypes and wireframes to the hi-fi prototypes and style guides developed in Figma.

4.2.1 Paper Prototypes

Paper prototypes are wireframes drawn on paper. This is usually the first step in user interface design. The process of drawing paper prototypes was started for MS24 but was quickly decided to be too time consuming, due to the high number of circles and small elements required. Figma was deemed a better option for this task, so the initial interface design was digitally created through lo-fi wireframe prototypes. The initial paper prototyping attempts have been included below.

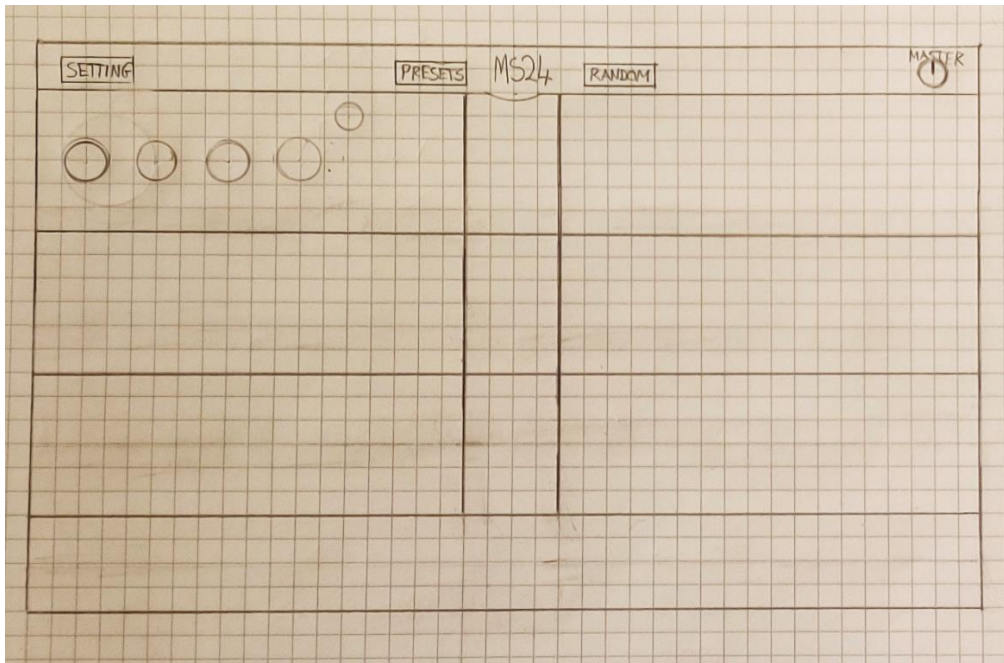


Figure 45. Paper prototyping attempt one.

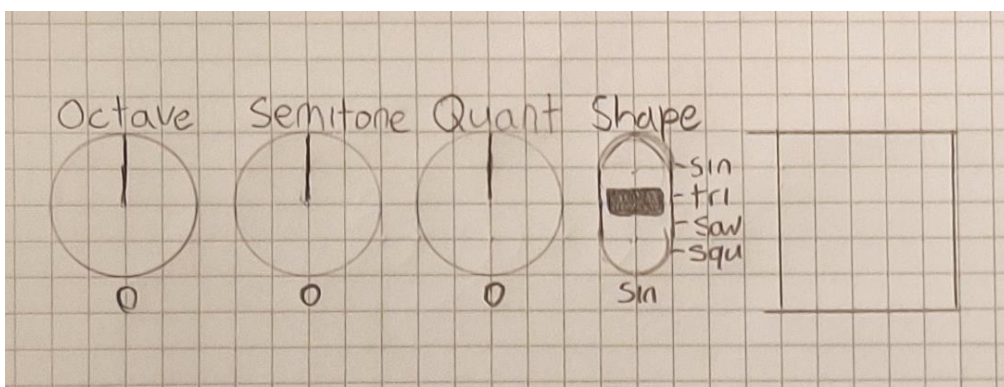


Figure 46. Paper prototyping attempt two.

4.2.2 Low Fidelity Prototype (Wireframe)

A lo-fi (low fidelity) prototype or “wireframe” for a webpage focuses on the basic layout of content, excluding colours and detail.

Since MS24 is to be a web synthesizer, a lot of detail went into the wireframe, as decisions about placement of controls and elements had to be made as soon as possible, since a large part of what distinguishes a good synthesizer from a bad one is the interface. By referencing Tone.js to see which properties were available for each of the major components of the synthesizer, the required number of controls in each group was determined before placing them in the most logical order according to the planned signal flow of the synthesizer, as seen in the flow charts made during process design (Figure 44). Popular synthesizer VSTs such as Serum and Diva were referenced throughout this process, however much of the design is entirely original as they were only used to get a rough idea of controls and positioning.

Wireframes and hi-fi prototypes were created using the free web-based design tool Figma, which functions similarly to Adobe Illustrator and Adobe XD.

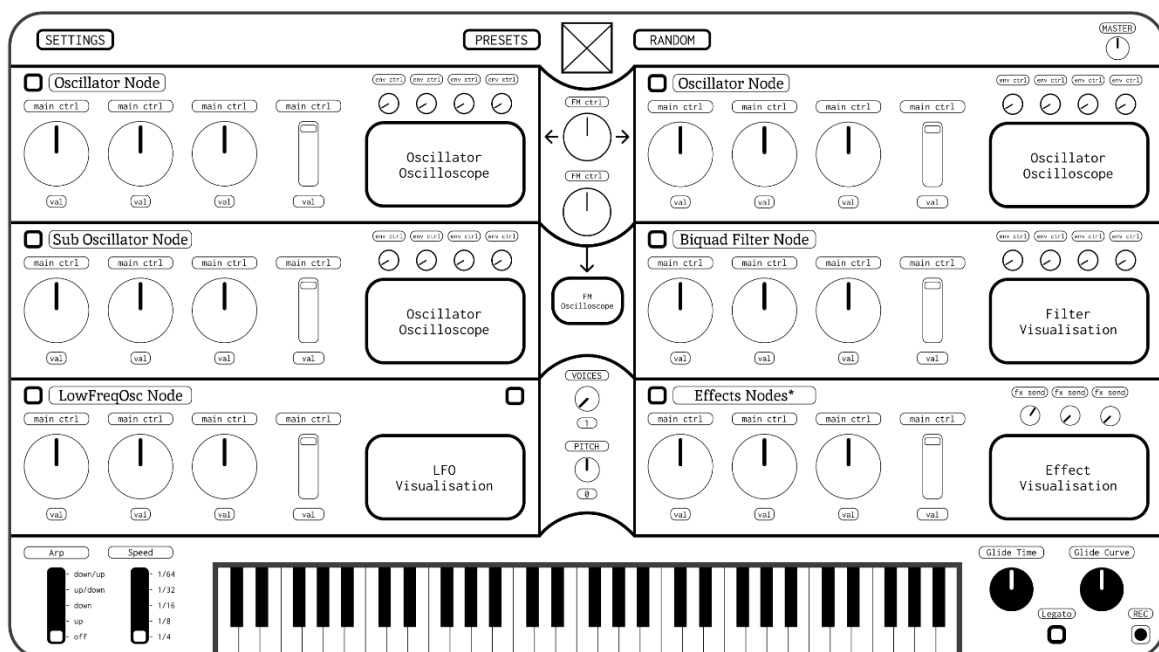


Figure 47. MS24's wireframe design.

4.2.3 High-Fidelity Prototypes

A hi-fi (high-fidelity) prototype is a more advanced and detailed version of a wireframe that incorporates visual design elements such as colours, typography, and images, as well as interactive features such as clickable buttons and forms. In MS24's case, the main difference between the wireframe and hi-fi prototypes is the inclusion of colour, the proposed logo for the synthesizer, and visualisations. A light and dark variant was designed, as well as a diagram showing the signal flow of the synthesizer, each of which are included below.

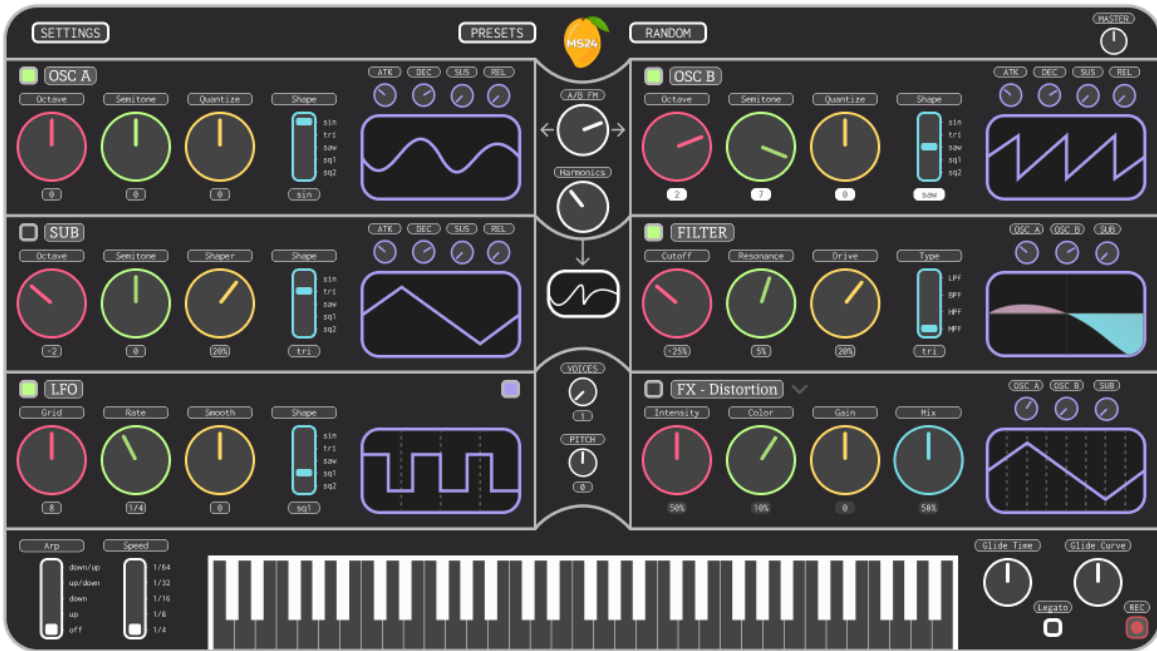


Figure 48. MS24 Hi-Fi Prototype (dark version)

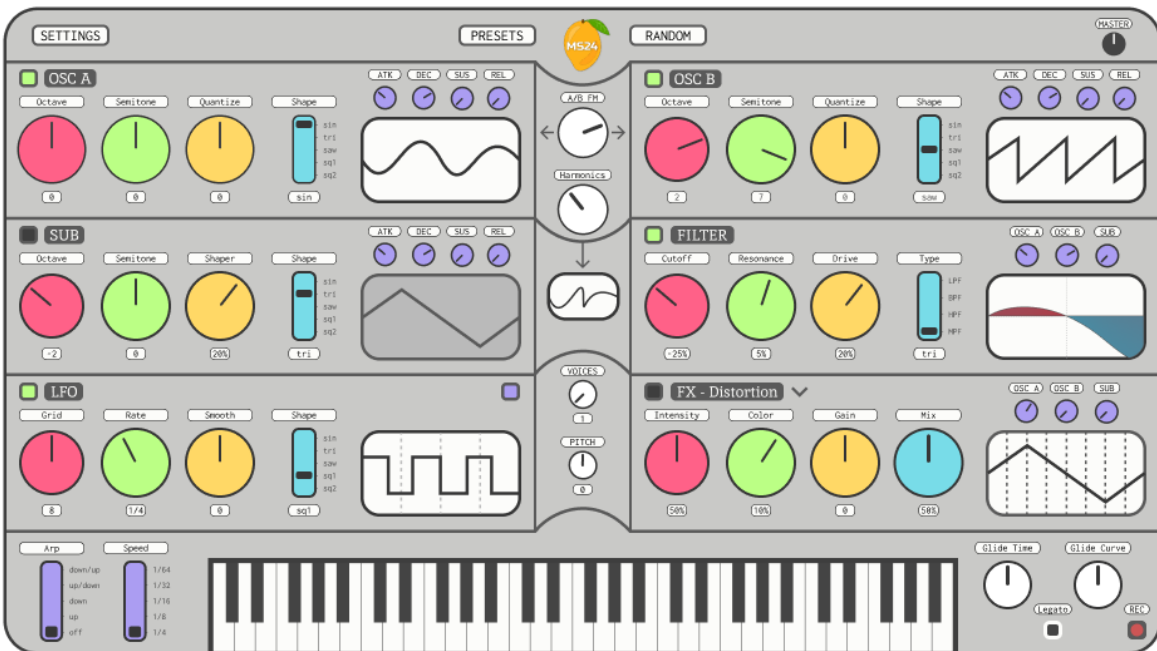


Figure 49. MS24 Hi-Fi Prototype (light version)

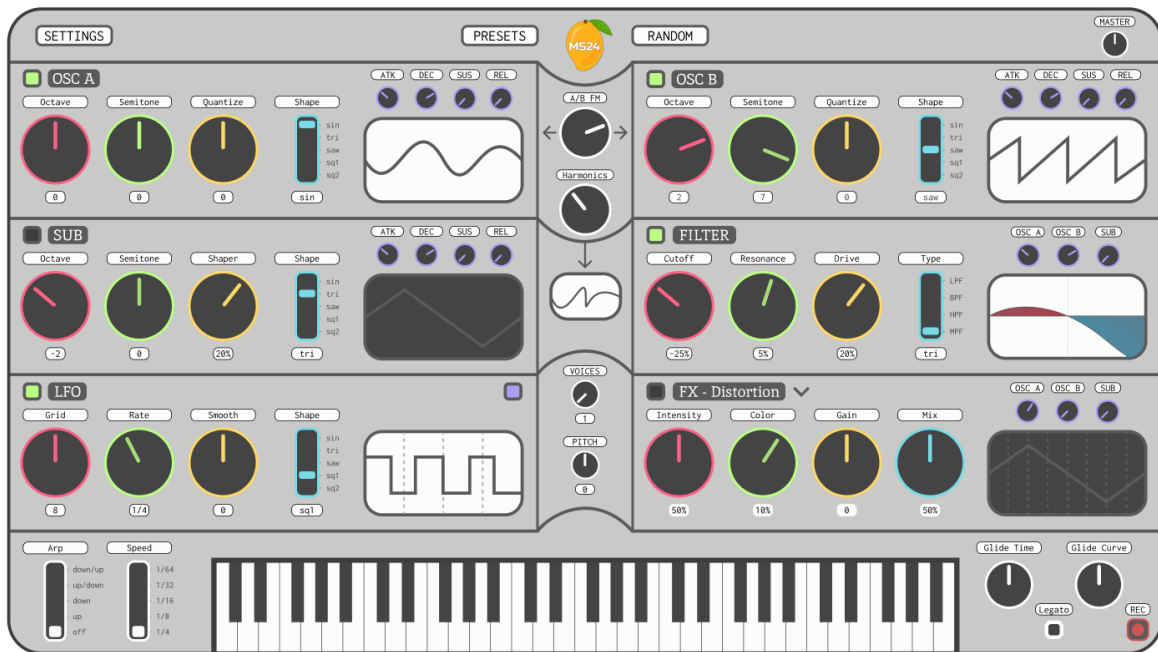


Figure 50. MS24 Hi-Fi Prototype (alternative light version)

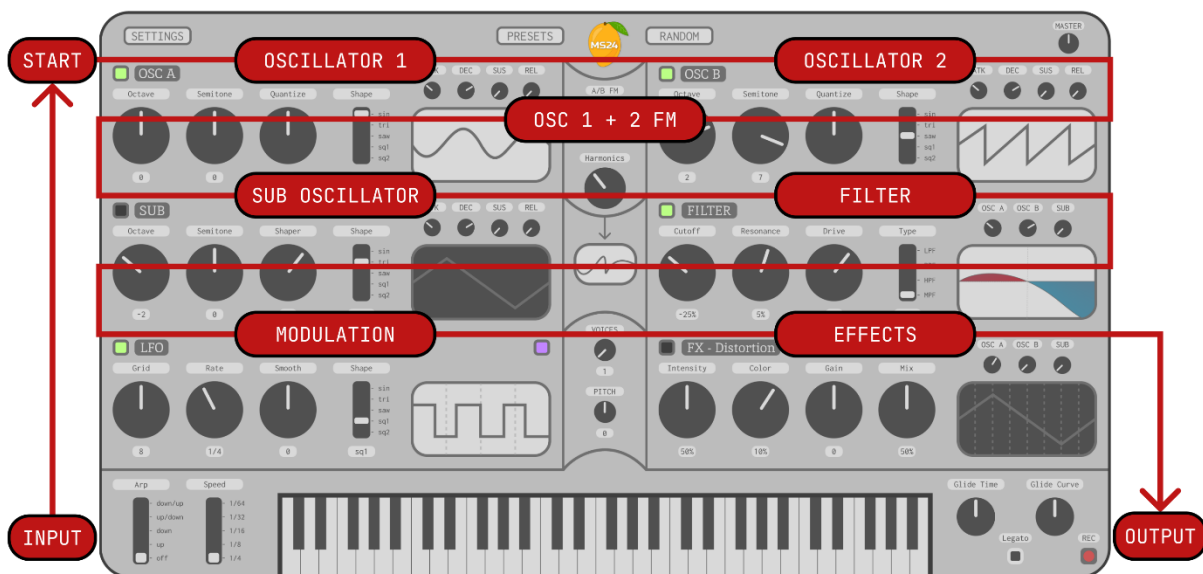


Figure 51. MS24 Hi-Fi Prototype (annotated version)

As can be seen in the image above (Figure 51), the major components of the synth were laid out in accordance with the signal flow, incorporating a zig-zag pattern as the signal travels from the top left of the interface to the bottom right. Another way this could have been laid out would have been to have each of the three oscillators on the left, with each of the signal modifiers on the right. However, this was decided against since the planned A/B frequency modulation (FM) controls were placed in the middle of the synth, making the zig-zag layout more intuitive, as oscillator 1 and 2 would be affecting each other as the signal flowed between the two.

Context menus and the presets page of the synthesizer were also designed during the high-fidelity prototyping stage, which have been included below.

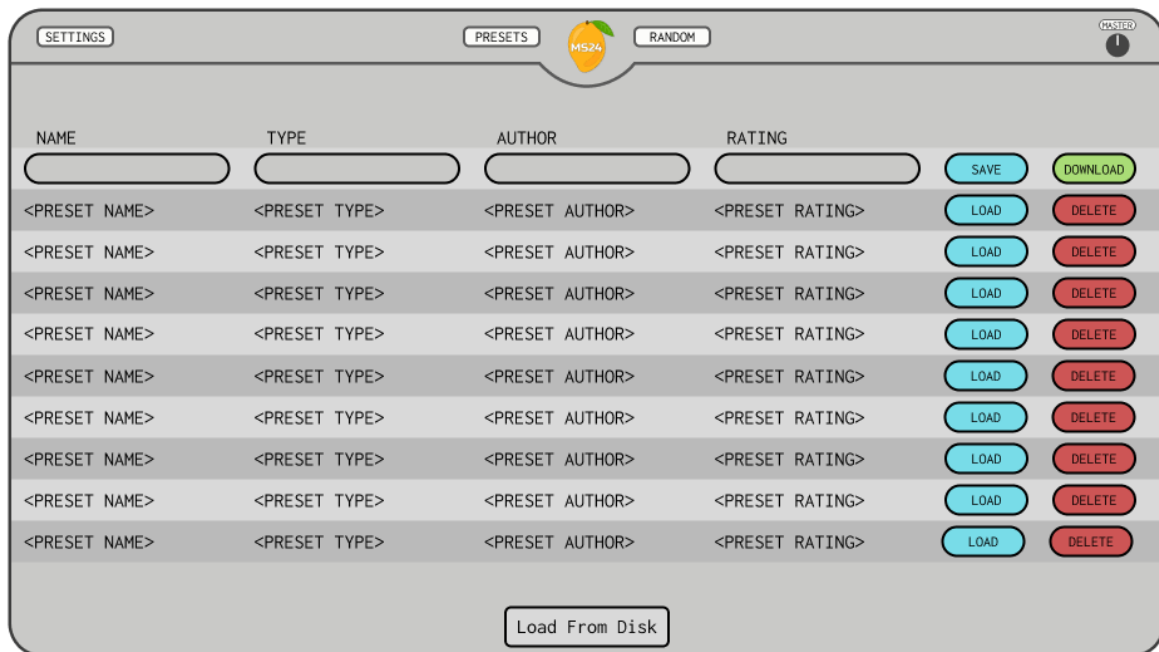


Figure 52. MS24 Hi-Fi Prototype (presets page)

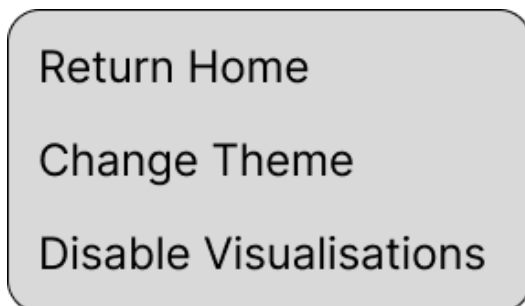


Figure 53. MS24 Hi-Fi Prototype (context menu)

4.2.4 Style guide

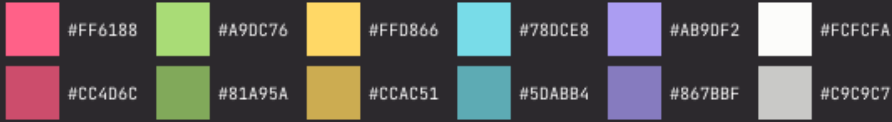
A style guide is a set of standards and guidelines that define the design elements and visual language to be used in an application. They typically include specifications for typography, colours, imagery, and layout, as well as guidelines for user interface components such as buttons and forms.

In the case of MS24, a style guide was created for both dark and light themes, including colour palettes, typography, and GUI elements such as the sliders, knobs, and switches made with the webaudio-controls JavaScript library.

MangoSynth24

UI Style Guide
Dark Version 1

Main Palette (Monokai)



Main Font (Inconsolata)

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
1 2 3 4 5 6 7 8 9

Regular
Bold

Secondary Font (Inika)

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
1 2 3 4 5 6 7 8 9

Regular
Bold

Sliders



Knobs



Switches



Scopes



Figure 54. MS24 style guide (dark version).

MangoSynth24
UI Style Guide
Light Version 1

Main Palette (Monokai)



Main Font (Inconsolata)

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
1 2 3 4 5 6 7 8 9

Regular
Bold

Secondary Font (Inika)

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
1 2 3 4 5 6 7 8 9

Regular
Bold



Figure 55. MS24 style guide (light version).

4.3 Conclusion

In conclusion, the Design chapter examined the design phase of MS24. During this phase, the program design was created, consisting of a plan for the technology stack, an examination of the structure of those technologies, programming paradigms and design patterns possible to incorporate with those technologies, the architecture of the application, being the combination of those technologies, and the process design, being the application's flow of logic. Following this, the user interface design was created, consisting of paper prototypes, wireframes, high fidelity prototypes, and style guides. This process sought to ensure a smooth implementation phase of MS24, through a solid plan of action, answering many of the initial development questions prior to the development process.

5 Implementation

This chapter describes the implementation of MS24 over the course of eight Sprints. The following technologies were used throughout:

- React

The first drafts of the application were built using the popular JavaScript framework React. This was to be used to componentize the code, and to allow for powerful state management if required.

- Zustand & Jotai

Two popular state management libraries were tested with React in the first drafts of the application. These libraries are used to simplify the process in various ways but were chosen mainly for the fact that they make it easier to manage global state, which would otherwise require “prop drilling”.

- ‘Vanilla’ JavaScript

Vanilla JavaScript was migrated to after it was decided that React was overcomplicating the development of the application with no real need to use it, due to the fact that most of the libraries used manage their own state, and since the application would reside on one page.

- HTML5

HTML5 is used to create the structure of the application’s frontend, mainly using `<div>` elements and custom GUI elements from the ‘webaudio-controls’ library.

- Tailwind CSS

Tailwind CSS will be used to style the application’s frontend, applying everything from the fonts that text uses to the position of elements on the webpage.

- Vite

Vite was chosen as the frontend build tool for the application, as it significantly improves the development experience by providing extremely fast hot reloads and optimized bundling. This was used in both the React and Vanilla versions of the application.

- Tone.js

This JavaScript library abstracts the Web Audio API, making it easier to work with in lots of ways. This library allows for the underlying synthesizer architecture to be developed in accordance with the design spec.

- webaudio-controls

This JavaScript library provides prebuilt custom HTML input elements such as knobs, sliders, and a keyboard, which is used to create the main GUI controls for the application. These controls have event listeners attached to them which will update Tone.js settings as they are changed.

5.1 Scrum Methodology

For the implementation phase of this project, the Scrum development framework was utilized to efficiently achieve each project goal. The framework draws its name from the way a rugby team trains for a big game, where a "scrum" is used to promote healthy critical analysis of the current situation, learning through experience, ensuring self-organization around every problem that arises, and reflection on both successes and failures to foster continuous improvement.

The Scrum framework is heuristic, which means that it is flexible and can adapt to the changing factors that arise during development. It recognizes that teams may not have all the necessary knowledge at the start of a project and that understanding evolves over time. Regular re-prioritization is a critical aspect of this framework that allows for continuous improvement. (Drumond, n.d.)

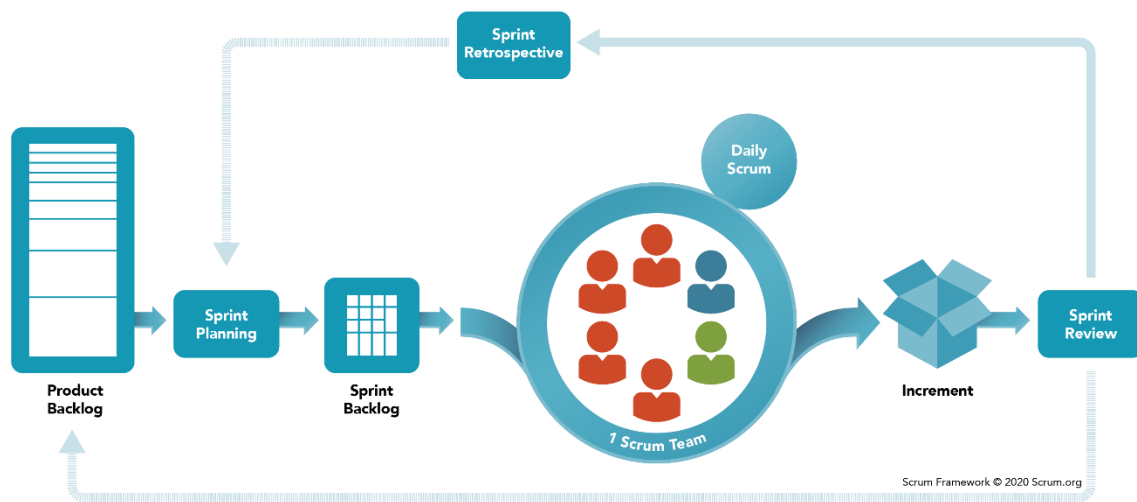


Figure 56. A diagram of the Scrum process (image retrieved from <https://www.scrum.org/resources/what-is-scrum>)

In Figure 56 above, the Scrum workflow is illustrated, highlighting several crucial stages. Firstly, the 'Product Backlog' contains a dynamic list of tasks that need to be completed, typically overseen by the product or service owner or manager. This serves as a to-do list for the entire team. The team then plans 'Sprints' - collections of tasks - and creates a 'Sprint Backlog' that lists the tasks to be completed during that Sprint.

However, these lists are dynamic and subject to change throughout the Sprint. If tasks are not completed by the end of the Sprint, they will likely be considered for the next one instead. The team works towards the 'Increment' or 'Sprint Goal', making up the Sprint Backlog. At the end of the Sprint, the team reviews what has been accomplished, holds a retrospective to analyse what worked and what didn't, and decides how to deal with uncompleted tasks in the following Sprint or Sprints. The team then returns to the Sprint Planning phase for the next Sprint, continuing this cycle until the Product Backlog is empty - which may or may not occur depending on the project's nature.

5.2 Development environment

5.2.1 WebStorm

WebStorm is a powerful Integrated Development Environment (IDE) developed by JetBrains. WebStorm was chosen to develop MS24 with as it is designed to facilitate the development of web applications and is highly optimized for web technologies. With its user-friendly interface and extensive feature set, WebStorm is an ideal tool for web developers seeking to increase their productivity and streamline their workflow.

WebStorm's intelligent code editor offers advanced code completion, error detection, and refactoring features that help developers write clean, concise code quickly. The IDE also includes integrated debugging and testing tools that enable developers to identify and fix issues in their code quickly.

Furthermore, WebStorm's integration with popular version control systems, such as Git and GitHub, allows for seamless collaboration among team members, improving the overall development process.

5.2.2 Git

Git is the version-control system that was used for MS24, through GitHub, the largest and most popular repository host in the world, with over 61 million users at the time of writing (retrieved by using GitHub's search engine, filtering by users)

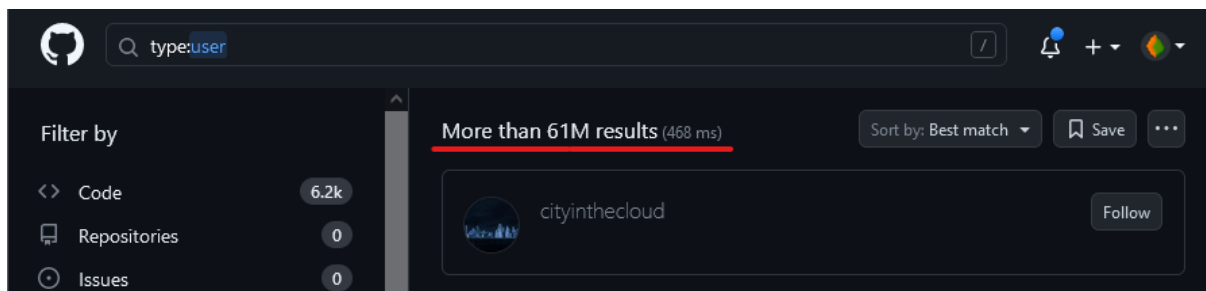


Figure 57. GitHub's user count at the time of writing.

The command-line interface (CLI) within WebStorm was primarily utilized to regularly execute git add and git commit commands. However, for the purpose of composing informative, multi-line commit messages, GitKraken was employed as an additional tool, which is available as part of the GitHub Student Developer Pack.

GitKraken provides a graphical user interface (GUI) that can perform all the actions that can be executed through Git and even includes its own custom CLI for advanced users. Prior to pushing changes to the remote repository, the repository tree was visualized in GitKraken, and commit messages were reviewed for formatting and spelling errors. This helped to ensure that all the appropriate changes were included in the commit, while preventing any unwanted changes from being inadvertently included.

5.3 Sprint 1

The goal for this sprint was to implement the literature review which analysed the feasibility of the browser as a computer music development platform, and to conduct further research on synthesizer architecture, before moving on to requirements gathering.

5.3.1 Item 1: Literature Review & Extra Research

Having previously written a literature review titled '*Feasibility of the Web Browser as a Platform for Interactive Audio Synthesis*' for the Research & Analytics module at the end of the first semester, it was time to incorporate it into the research chapter of this report. However, some extra context was deemed necessary, to give the reader a better understanding of basic acoustics and common components found in synthesizers. Extra literature was read, and relevant information was referenced above the literature review.

5.3.2 Item 1: Prototyping – Paper Prototypes

Paper prototypes were planned to be developed prior to lo-fi wireframes and hi-fi prototypes to get an idea of the interface design. However, due to the number of circles and curves required, it turned out to be too difficult, especially with a compass that refused to lock once set! It was quickly decided that Figma would be a lot easier to use for the initial interface design. Photos of the attempts are included in the Design chapter (Figures 45 and 46).

5.3.3 Item 2: Prototyping – Wireframes & Prototypes

Wireframes and prototypes were developed using Figma, the free vector-based prototyping tool which offers an experience like Adobe Illustrator and Adobe XD combined. While the designs created are original, they took inspiration from various virtual synthesizers I've used over the years, such as Serum and Diva.

5.4 Sprint 2

The goal for this sprint was to gather requirements by looking at similar applications and by making a survey, as well as to develop the design document with improvements to the prototypes and to get a start on the planned design through diagrams and pseudocode.

5.4.1 Item 1: Requirements Gathering

After looking at the initial work done for requirements gathering during the first semester, it was decided that similar applications which more closely relate to the web-based application should be examined, and a survey should be conducted so that personas could be created to determine the functional and non-functional requirements of the application.

5.4.2 Item 3: Prototype Improvements

Some improvements were made to the hi-fi prototypes during this sprint. The colour scheme was updated, and the design of the controls were changed to closer match what the JavaScript library 'webaudio-controls' could create. Menus & alternative FX units were also added.

5.4.3 Item 4: Google Forms Survey

A survey was created using Google Forms and posted to several Discord servers. The data would later be compiled, and the requirements would be updated, however this did not happen during this sprint as the survey took a while for responses to build up to a substantial number.

5.4.4 Item 5: Design Document

The design document was started during this Sprint, with diagrams made and pseudocode written to plan the internal structure of the application ahead of time.

5.5 Sprint 3

The goal for this Sprint was to begin developing the application prototype using the research, requirements, and design documents to inform decisions made around the code. Another goal was to update any previous documents where necessary, and to begin filling out the implementation chapter.

5.5.1 Item 1: Initial Development – Application Structure

The development of MangoSynth began in this Sprint, which took place in WebStorm using React initially. Previous prototypes of the application had been written in pure JavaScript and P5.js, but after a supervisor meeting it was deemed a good idea to use React in case of user state issues down the line, if trying to store users' configurations using a database and server.

To begin with, the structure of the React components had to be figured out. Which file would contain the main Tone.js logic? Could Tone.js be componentized so that individual parts of the synthesizer could be reused to cut down on the package size?

While developing the first prototype, it was discovered that componentizing Tone.js would be a difficult and confusing task, due to the nature of React and state management. Event listeners set up in one file would need to talk to the other files in perfect sync, to initiate Tone.js triggers so that the right sound was created. It seemed like it might only be possible to do this in a single file, using class components instead.

5.5.2 Item 2: Initial Development – React, Vite, Zustand, Immer

A second prototype was started after the discoveries made during the first warranted a new approach. Vite was installed and a React project was initialised using it to cut down on boilerplate and speed up development builds. A far simpler component hierarchy was built, with a single component called Synth.jsx which included all the Tone logic, as well as the event listeners required to use the GUI library. Many different state management strategies were then attempted, including default React state functionality, importing the Zustand and Jotai state management libraries. Zustand was found to be the best for the job, allowing for the synthesizer settings, GUI controls, and application data to be contained in easily configured state factories, with getters and setters available throughout each layer of scope.

This new approach worked a lot better than the last, and still allowed for the separation of the major parts of the synthesizer through use of React class components. However, the GUI was still a little bit buggy due to the use of useEffect() React hooks and certain keyboard logic such as handling when to fire attack and release signals through Tone would break if forced to be stateful.

After going through these issues with the project supervisor, the question was asked: “What is React even doing by this stage?”. So far, state management and componentization had proven to be the cruxes of the issues encountered, and it was possible neither of these things were even necessary for the application. It was suggested that it might make more sense to rewrite the code in Vanilla JavaScript instead, which could lead to a less buggy GUI and a more manageable code structure.

5.5.3 Item 3: Update of Previous Documents

The previous chapters of this document were worked on more during this sprint. The various diagrams and wireframes were imported into the design document, and some of the application architecture documentation was updated.

5.5.4 Item 4: Implementation Chapter

This chapter was started during this Sprint. This involved referring to the backlog to piece together what had been done and in what sequence.

5.6 Sprint 4

The primary goal of this Sprint was to migrate the existing React-based architecture to a Vanilla JavaScript based architecture. The decision to switch to Vanilla was made during a Sprint meeting with the Scrum master, after it was concluded that React was likely overcomplicating things, with no real reason to use it. The design documentation would also need to be updated to reflect the changes made.

5.6.1 Item 1: Migrating to Vanilla

To begin migrating from the React prototype to Vanilla JavaScript, the key functionality of the existing implementation had to be identified so that the reimplementing with pure JavaScript

syntax could go ahead smoothly. Initially, a new project was created in WebStorm using the HTML5 Boilerplate template provided by JetBrains. This template includes a vast number of optional files to help with starting a Vanilla web application, however all but *normalize.css* was deleted, which JetBrains obtains from a popular repository on GitHub under the same name, which contains a CSS file that normalizes styling across all browsers.

After getting set up and ready to start developing, it was realized that a build tool would need to be used if ToneJS was to work, as the library is only provided as an NPM package. If the project didn't require any packages, when it came to releasing a production build, only the index.html file alongside its dependencies would need to be hosted. Since it did, the project files needed to be "bundled" by a build tool, to combine each of the dependencies into single, optimized HTML, CSS, and JS files.

Vite was chosen again, since it had been previously picked during the React implementation due to its extremely high popularity, as found on the [State of JavaScript](#) website, which has been collecting data from over 20,000 developers each year since 2016 to identify trends. Vite is useful as it provides a bunch of templates for getting started with different JavaScript architectures, including one for Vanilla projects. However, it was decided at this point to use Tailwind CSS for styling, as it is a popular CSS framework packed with helper classes, allowing for quick inline styling with well-tuned defaults.

After researching how to use Tailwind with Vite, it was found to be a bit complicated. It was unclear whether it was worth the effort until a link to a repository in the Vite documentation was remembered, which contained a list of community Vite templates. This repository contained two templates for a Vanilla JS project using Tailwind. The first of these, [template-vite-vanilla-tailwind-v3](#), was last updated in March 2022 and contained outdated versions of the dependencies, but the second, [Vitawind](#), was up to date and provided a simple to use template to get started with.

The reimplementing of synthesizer logic went surprisingly well. This was because most React features weren't being used during its implementation, with code written that would normally only be seen in Vanilla JS, such as event listeners attached to HTML elements, as this was required to use the webaudio-controls API.

Most of the logic was easy to port. The main difference was that there was less of a need to battle with multiple state managers, so more efficient and simple code for event listening could be written. The GUI was also developed more smartly than in the previous prototypes. A bonus of not fighting with state management anymore was that the webaudio-controls library no longer glitched visually when changing parameters. It quickly became apparent that trying to use React and force the library to be stateful really had been a mistake.

5.6.2 Item 2: Tone.js Logic

After the migration, more Tone.js synthesizer logic was worked on. Three Tone PolySynth objects were added, each using an FMSynth as the voice generator. PolySynth is a special Tone synthesizer which allows polyphony. Polyphony is the ability for a synth to play multiple notes, or "voices", at once. A "voice" in synthesizers means an individual note playing, so in a monophonic synth, if one were to hold down a key on their keyboard to play a C note, and while still holding C, played a D note, the synth would immediately switch to playing D, and C would cut out. Chords are impossible

with monophonic synthesizers, so using PolySynth objects for the oscillator groups would be the only way to avoid writing custom polyphony logic, which could be quite difficult.

A PolySynth takes two parameters, <voice> and <options>. The voice parameter looks for one of Tone's synth types, such as an FMSynth, which it creates a new instance of for each key held down, to achieve polyphony. The three PolySynths added would act as OSC A, B, and C. They were then connected each of these synth objects to a filter, and a distortion effect. With the ported keyboard logic and the webaudio-controls GUI connected, a lot of different sounds could now be created.

That's all that was added to the synth during Sprint 4. During Sprint 5, multiple FX units are planned to be added, which users will be able to switch between. Modulation of parameters through an LFO, glide (portamento) and unison (voice harmony offset) controls will also be investigated.

5.7 Sprint 5

The goals for Sprint 5 were to present the application prototype to the project supervisor and second reader, as well as to further develop the application, after prioritising the features that remained undeveloped. The interim presentation gave insight into this, and it was decided that the ability to save and load "presets"; configurations of the synth, should be prioritized, as it would make the application feel a lot more complete if it shipped with a list of presets for users to try out, as this is what mostly all virtual analogue synthesizers do.

5.7.1 Item 1: Toggleable Synth Groups

Prior to the interim presentation, the application prototype only had a basic synthesizer interface built, with only three oscillators feeding into a filter. The LFO modulation hadn't been implemented, and there was only one FX unit, the distortion effect. Additionally, there was no ability to toggle each synthesizer unit on or off. It was felt that this was a bit lacklustre, so it was time to implement some changes.

This update was started by adding the ability to toggle units on and off, removing them from the Tone.js signal chain when turned off, and adding them back in when turned on. This wasn't as simple as it might sound, as the nodularity of the Web Audio API was a little confusing at first. Originally, the main sound generators (the oscillators) were connected directly to the master output (Tone's destination node). The master node was then connected to the filter and distortion effect in series.

Using Tone's disconnect method on the oscillators worked fine, as this simply detached the oscillators from the output. When it came to disconnecting the filter and distortion however, using the disconnect method on these nodes wouldn't work, and disconnecting the destination node seemed to irrecoverably break the signal chain.

After delving into the Tone.js documentation, it became apparent that disconnect completely removes a node from the chain, and in the case of the destination node, simply re-initialising it by writing "Output = new Tone.Destination" wouldn't work.

To fix this issue, the signal chain had to be rethought.

Instead of it being *Oscillator a/b/c -> output -> filter -> effects*,

It had to become *Oscillator a/b/c -> filter -> effects -> output*.

This way, each component on the chain could be disconnected without impacting the destination node, which is responsible for outputting the sounds generated to the user's default playback device.

With the filter toggle now working, it was discovered that the distortion effect presented a new challenge. Disconnecting and reconnecting an effect in Tone did not seem to work, but luckily the effect nodes each have a "wet" property. The wet property determines how much of the signal is returned effected. A "dry" signal is one with no effects applied, a half dry, half wet signal is one in which you can hear half of the original signal and half of the effected signal, and a completely wet signal includes none of the original signal, unless of course the effect is turned down. Despite being a somewhat unconventional solution, using the wet property seemed to be the only option for toggling the effects. Instead of removing the effect from the signal chain, the wet signal was simply silenced. A new variable was created to keep track of the previous mix value, as each effect had a "mix" knob (or "dry/wet" knob), and it was important to know what value to return the wet property to when toggling the effect on. If the user changed the mix knob while the effect was off, the value would update accordingly so that toggling the effect on would set it to the correct value.

5.7.2 Item 2: FX Switching

When implementing new effects, initialising the ones provided by Tone.js was straightforward, but the challenge was in switching between them and updating the GUI to reflect the available controls. Not all effects had four parameters, but there were four knobs available. A function was required to dynamically update the webaudio-controls elements to cleanly implement an effects selector.

The HTML was first updated to include a select box with options for each effect, and a JavaScript event listener was created to listen for changes made to the select box to update the GUI accordingly. Referring to the webaudio-controls documentation, it appeared that updating the GUI after the initial render was only possible through some unfamiliar-looking jQuery syntax. However, using the same syntax with the properties for values, min/max, and step increments also worked. Importing a version of jQuery from a CDN was required, which was not mentioned in the documentation, but this was discovered after Googling errors that were logged to the console and finding an answer on StackOverflow.

As there was seemingly no way to cleanly split the syntax into multiple lines in the HTML, resulting in lengthy and hard-to-read code, the decision was made to attempt using the syntax in the synth.js script instead. After carefully consulting the Tone.js documentation to determine the appropriate minimum and maximum values for each effect parameter, as well as the best default values, a large if/else tree was created to decide which GUI knobs to re-render or remove when selecting a new effect. The old signal chain was then replaced with the new one, resulting in a total of nine hot swappable effects.

5.7.3 Item 3: LFO Modulation

Next, LFO (low-frequency oscillator) modulation was introduced. The theory behind it was straightforward: limit an oscillator's frequency to a very low range (0.001Hz to 5Hz), and then use its output to modulate another value. This modulation can add a sense of movement to sound design, like gradually increasing and decreasing the cutoff frequency of a filter.

The implementation of LFO modulation turned out to be simple when connecting it to a parameter, but the challenge was switching the modulation target. Disconnecting the LFO from a target resulted in the target no longer functioning, causing the filter to break even after reinitializing it. This was like the issue with the original effects implementation and the destination node, but no solution was found after spending several hours trying different syntax, referring to the documentation, and looking at various GitHub issue discussions in the Tone.js repository. As a result, the LFO could only modulate the filter cutoff for now.

5.7.4 Item 4: Recorder, Missing Controls, Interface Update, MIDI

There were still some tasks left to be completed before the interim presentation, so they were started on. These tasks were smaller ones, but they would make the application feel more polished.

A recording feature was implemented, with a toggle button added in the bottom right corner of the interface. When the toggle button was turned on, a Tone.js Recorder object would start listening to the output of the synth. Once turned off, the recording would be downloaded to the user's computer as a file with a ".webm" filename extension, following the Tone.js documentation. However, upon playback with foobar2000, it was discovered that the file was actually an Ogg Vorbis container with a ".ogg" extension. Despite thinking that the filename output would determine the encoding type, it only determined the filename. It would have been nice to be able to record in a lossless format such as WAV or FLAC but there didn't seem to be a native way to do so.

Upon researching the issue, it was discovered that the MediaRecorder API used by the Recorder object in Tone.js only supports the *audio/ogg* codec and does not even allow for high quality, lossy MP3 files to be output. A StackOverflow post suggested that it was possible to expand the API's codec output capabilities, but the process seemed complex and would require the creation of a custom Recorder functionality, as well as a library such as "libflacjs" to correctly encode the audio stream.

Although the current implementation of the recorder would suffice for the time being, it was acknowledged that this limitation would need to be addressed later to make the synth more suitable for music producers. The use of lossy formats was deemed undesirable in music production due to the potential loss of audio fidelity upon transcoding to a lossy format, which can occur when uploading music which uses lossy audio samples to various platforms like YouTube, Spotify, or SoundCloud, due to repeated compression.

Next, a master gain control was added to the synthesizer interface. It was placed in the top-left corner of the interface, giving users the ability to adjust the overall volume of the synthesizer. This was useful for making quieter patches more audible, or for preventing loud patches from clipping.

Envelope control for each of the oscillators was then added to allow users to change the attack, decay, sustain, and release values for each sound generator. These controls are useful for changing the characteristics of a sound. The attack determines how quickly a sound reaches its maximum

amplitude. The decay determines how quickly the sound drops from maximum amplitude to the sustain level. The sustain level determines the volume that the sound rests at before it is released. The release value determines how quickly the amplitude drops from the sustain level to silence.

A high attack and low release would mean that when a key is held, the sound gradually fades in before abruptly stopping once the key was released. This could be used to emulate, for example, a violinist slowly bowing the strings of a violin, gradually increasing pressure on the strings before releasing the bow entirely and muting the strings. A low attack value with a high release, on the other hand, could be used to emulate a pianist striking a key with a lot of force while holding the sustain pedal down with their foot. This would result in a sharp note followed by a long resonance as the string naturally loses vibration before eventually stopping entirely.

The proposed design of the synthesizer interface positioned the ADSR envelope knobs above the visualisers. HTML and CSS code for those scopes were written at the same time as they required a bit of Flexbox magic. A canvas element was put in each of the visualisation boxes, and the visualisation logic was left for after the presentation as it would require more time to figure out.

After updating the interface so that it more closely matched the proposed design, it was decided to investigate how webaudio-controls integrated with MIDI keyboards. The Akai MPK Mini MK3 was used for this experiment. While it was known that the library provided some integrations out of the box, it wasn't clear how well these worked. Surprisingly, the MIDI learn functionality worked a charm, allowing any of the knobs or sliders to be right clicked, bringing up a context menu with three options: "learn", "clear" and "close".

Upon clicking learn, it would start listening for a MIDI control. By turning one of the physical knobs on the MIDI keyboard, it took control of the GUI knob that had been selected. It was now possible to assign each of the knobs on the keyboard to any control on the synth, allowing them to be changed while keys were being held on the computer keyboard.

Sadly, the MIDI keyboard's keys didn't automatically assign themselves to the webaudio-controls keyboard, so this would have to be figured out later. The only issue encountered was that if a control was right clicked and the clear option was selected, it would only clear the assigned physical knob until the page was reloaded. Upon page reload, the cleared controls would be re-learnt! This was fixed by turning off a global webaudio-controls setting called "preserveMidiLearn" which was intended to store learnt controls in the browser's memory. It would have been nice if it remembered what had been cleared, so that someone could keep all the learnt MIDI controls between sessions, but sadly this wasn't possible.

5.7.5 [Item 5: Hosting](#)

Hosting the application was briefly attempted while the application was being built with React. The reason for attempting hosting so early was mostly due to prior experience with issues when hosting. However, this attempt didn't get very far, as the repository being used for MS24's code is an organisation-managed repository from IADT. Despite a Vercel access request being sent in Sprint 2 or 3, it had not been accepted upon checking. Before trying again with the new Vanilla architecture, an email was sent to the member of staff responsible for managing the organisation repos to ask if the request could be accepted. Once the request had been accepted, the deployment process was a breeze, requiring only a couple of clicks before the app was automatically built and hosted online.

The only issue encountered was that the hosted version of the application had a problem with the effects selector logic. Everything else worked fine. At first it seemed as if the jQuery logic used for manipulating the webaudio-controls GUI elements wasn't going to work online, but upon checking the code, it was found that an old CDN URL for jQuery was being imported, which was a HTTP URL rather than a secure HTTPS URL. It's likely that Vercel blocked this HTTP URL for security reasons. After updating the URL to the latest version of jQuery with a HTTPS URL, the issue was resolved.

5.7.6 Item 6: Interim Presentation

After the updates to the application, a presentation needed to be prepared for the interim presentation with the supervisor and second reader. The presentation was created using Canva instead of PowerPoint, as it provides many great looking presets, especially with Canva Pro, which is provided by the GitHub Student Developer Pack.

Seven slides were created, which included an explanation of the application, a research slide, a technology slide, an implementation slide, and three slides showcasing the proposed designs.

The presentation was given via Microsoft Teams, and after the slides, a demo of the hosted application was presented with an explanation of each component. Feedback was received, and both lecturers suggested prioritising the presets functionality going forward. Overall, the presentation went well.

5.7.7 Item 7: MS24 Help Screen

After the presentation, the hosted application was shared with friends and family to showcase the progress made so far. However, it was realized that the basic controls, hotkeys, and keyboard controls were not explained.

To resolve this, the welcome screen was updated with instructions for each control type and the various hotkeys that could be used. Two text boxes were added to warn of existing bugs and how to avoid them, and the visuals were improved by adding a logo and updating the spacing between elements. This update greatly enhanced the welcome screen and made it more user-friendly, especially for those with less experience in using synthesizers.

5.7.8 Item 8: Synth Preset Functionality

After receiving feedback from the presentation, work was started on the presets functionality of the synthesizer. This required several commits to implement, as the goal was to gradually add the feature without compromising the functionality of the hosted application.

At the beginning of the implementation of the presets functionality for the synth, two large objects were created in the synth.js file. The first object included all the adjustable parameters of the synthesizer, acting as a reflection of its state at any given time. This would enable the object to be exported as a preset configuration that could be loaded later, reloading the adjusted parameters. The second object consisted of the minimum and maximum values for each adjustable parameter, which would help prepare for the future feature of randomizing the synthesizer configuration. Knowing the ranges of each parameter was crucial for this feature, as it would avoid the configuration from breaking.

After creating these objects, it was time to make the preset object stateful, so that any parameter adjustments would be reflected by it. The data structure was also adjusted to match the Tone.js logic, where oscillator A's shape would be "sine" instead of "O", eliminating the need for several variable maps. Remaining variable maps were then optimised and renamed, and some of the Tone logic was modified to read directly from the preset object, reducing the amount of redundant code.

To improve the functionality of the synth, a dropdown menu for presets was added to the interface, and the ability to save and load presets was implemented. When the save button was clicked, the presets object was downloaded as a "preset.json" file, which reflected the state of the synthesizer at that moment. Loading a preset required a function to update every GUI knob on the synth, as well as the readout underneath. Although time-consuming, this was achieved, and every knob in the synth would now update when a previously saved preset was loaded. The challenge came when attempting to load a non-default effect, which required the effect selector logic to be made into a function so that the knob min/max values, step increments, and available knobs in the effects group would also be adjusted when loading a preset.

At this point, loading a preset would update the GUI to match the state that had been saved, but it didn't change the sounds the synth made, as the Tone logic wasn't being updated yet. To do this, Tone setter syntax was added to the load function, so that each of the nodes would have their parameters set using the preset that had been loaded. A function was then created out of the Tone connections logic, so that the internal signal routing could be re-connected once each of the parameters had been set. Prior to connecting again, disconnect() had to be called for each of the synthesizer elements, as calling connect() without first disconnecting can sometimes duplicate connection rather than replace them.

It was now possible to create a synthesizer configuration, save it to disk, and load it again later. This ability was a significant achievement for MS24, as it would allow users to store and share their creations with others. Additionally, this feature enabled the inclusion of default configurations for beginners to experiment with and learn from. Sending a preset file to a friend for testing on the hosted version of the application proved successful and demonstrated the practicality of the feature.

5.7.9 Item 9: Code Clean-up

At this stage of development, the code for the synth was getting quite large and difficult to manage, with several, large branching if/else statements. To clean things up, these if/else statements were converted into switch statements, which helped to make the code more readable and maintainable.

5.8 Sprint 6

The goals for Sprint 6 were to continue adding any planned features that were missing, such as FM controls, unison controls, and oscilloscopes (waveform visualizers). Another goal was to find people to help with user testing, as the application isn't very suitable for unit testing, given that it's a synthesizer, and due to the uncommunicative nature of the Web Audio API.

5.8.1 Item 1: FM, AM, and Unison controls

From the beginning, MS24 was intended to provide a few powerful synthesis techniques which vastly improve the versatility of synthesizers that incorporate them. The first of these techniques is known as frequency modulation, or FM for short. As explained in the research chapter, FM alters the oscillating frequency of an oscillator, which is known as the carrier frequency, with another oscillator oscillating at a different frequency, known as the modulation frequency. This produces interesting wave shaping effects, heavily influencing the characteristics of an otherwise basic sounding waveform, such as a sine wave.

While initially designing the interface, FM controls were placed between oscillator A and B, in the middle of the synth. This was in part inspired by Serum, a popular virtual subtractive synthesizer developed by Xfer, which has two main oscillators, each providing the ability to modulate the frequency of the other, allowing either oscillator to be the carrier or the modulator. The problem with incorporating this design in MS24 was that all Tone.js' oscillator and synthesizer objects are triggered with a method that isn't compatible with live frequency adjustment. When triggering one of these objects using `triggerAttack`, the key or frequency is passed in, causing an oscillation to begin using the given value. To stop the oscillation, a method called `triggerRelease` is used, using the same key or frequency as before. To implement Serum's version of FM, the oscillating frequency of a synth object would need to be adjusted on the fly by another oscillator, but this just isn't possible with Tone.

Up until this point, each of MS24's oscillators were using Tone's `PolySynth` object, which provides polyphony control, taking any other Tone synthesizer object as an input, which it uses for each voice it manages. The synthesizer selected for each of the three `PolySynth` objects at this time was `Tone.Synth`, the most basic of the synthesizers provided, simply composed of a `Tone.OmniOscillator` fed into a `Tone.AmplitudeEnvelope`. After realising that the original idea for FM synthesis would be impossible, an experiment was conducted to see if it could be achieved per-oscillator in MS24, by setting the `PolySynth` objects voice generators to `Tone.FMSynth`, rather than `Tone.Synth`.

Tone's `FMSynth` is composed of two `Synth` objects, where one modulates the frequency of the other. `FMSynths` provide properties used to control the modulation, such as "modulationIndex" and "harmonicity", which unlike the `Synth` object's frequency, can be updated in real-time.

The main difference with using `FMSynth` objects to control frequency modulation instead of creating a custom implementation was that each oscillator group on MS24 would have their own FM controls, rather than the FM controls being independent of the oscillators.

While moving away from the original design seemed unsatisfactory at first, it was realised that this might improve the versatility of the synth much more, since each oscillator group could produce their own heavily modulated waveforms, which when combined could create many more unique sounds than if only one modulated waveform was output.

To begin implementing this, the three `PolySynth` objects used for each oscillator group had to be updated so that they used instances of `FMSynth` objects rather than `Synth` objects. The setter methods used also had to be updated to include the extra parameters, `harmonicity`, `modulationIndex`, and `modulationType`.

Harmonicity is described in Tone.js' documentation as being "the ratio between the two voices", meaning a harmonicity value of 1 would result in no change, whereas a harmonicity of 2 would result in an octave of difference between the carrier and modulator frequencies. `ModulationIndex`

on the other hand, is described as “essentially the depth or amount of modulation”, the ratio between the modulator’s frequency and amplitude. Modulation type refers to the basic waveform used by the modulator, such as a sine or triangle wave.

While reading the documentation for these properties, and after experimenting with the demos provided, it was noticed that there were some waveform shapes available that hadn’t yet been tested. Upon further inspection of the OmniOscillator documentation, since this was the object being used at the base level by the Synth objects, the information provided for the “type” parameter states that it can be set to any of the basic shapes or can be set to any of the basic shapes prefixed with “fm”, “am”, or “fat” to use the FMOscillator, AMOscillator, or FatOscillator types.

After plugging these values into the oscillator on the [FMSynth example](#) provided by Tone to see what they did, it was discovered that two new parameters appeared when prefixing a basic shape name with “fat”. These parameters were “count” and “spread”.

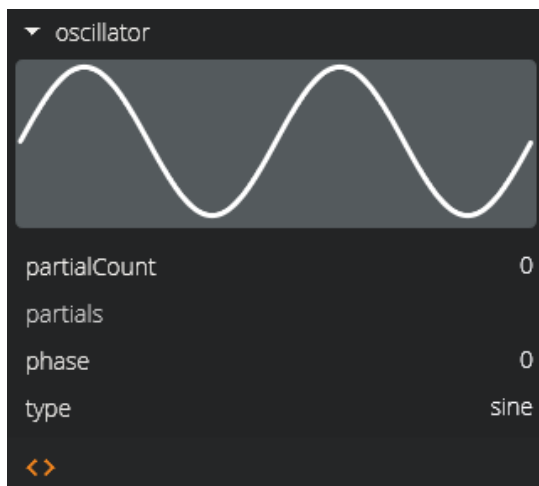


Figure 58. Tone's FMSynth example, using the default values.

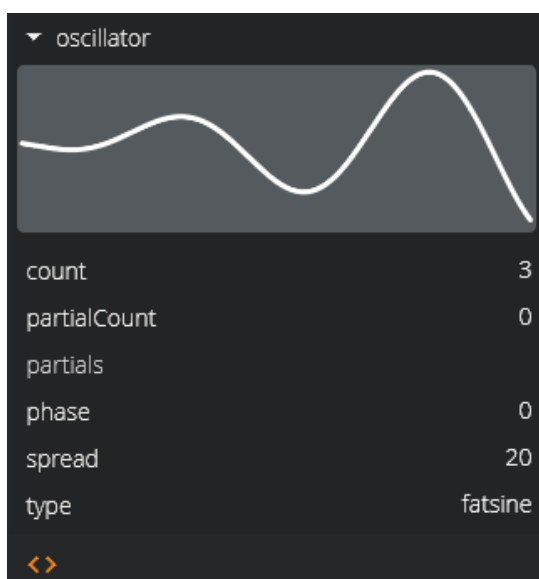


Figure 59. Tone's FMSynth example, with shape changed from "sine" to "fatsine".

To see what these parameters were exactly, the documentation for Tone's FatOscillator object was referred to. This documentation stated that a FatOscillator "is an array of oscillators with detune spread between the oscillators", meaning that this was how to incorporate unison with Tone.

Unison is another synthesis technique which was planned for MS24, as it is found in numerous VSTs and can be used to either "thicken" the sounds generated by an oscillator, or to introduce natural harmonic flanging effects, which would otherwise require an effect called a flanger, which isn't an effect provided by Tone.js.

Unison does this by multiplying the voices produced by an oscillator, and then offsetting the pitch of each new voice by a set amount, usually in cents, which are hundredths of a semitone. For example, if unison's voice count was set to 5, and the detune of each voice was set to 10, you would end up with five tones playing, one at -20 cents, one at -10 cents, one equal to the original frequency, one at +10 cents, and one at +20 cents. This is why the detune property is often called "spread", as it spreads the detuning effect out from the original voice in either direction.

After realising that unison could be implemented as well as frequency modulation, some more adjustments needed to be made to the PolySynth initialisation in the code, as well as the shape values set by the GUI sliders responsible. Each PolySynth setter would now also include a "count" and "spread" property to control the unison, alongside each of the FM controls explained earlier.

The value maps used to convert the numeric values sent from the slider controls into the corresponding shape (i.e., 0 would equal "sine") were then updated so that each of the available shapes were prefixed with "fat", meaning FatOscillator objects would be used by the two Synth objects which each FMSynth object consisted of, instead of the default OmniOscillator objects.

```



// Initialize PolySynth Object using Tone.Synth
const SYNTH_A = new Tone.PolySynth(Tone.Synth)
// Set PolySynth parameters
SYNTH_A.set({
  // Oscillator parameters
  oscillator: {
    // Oscillator shape
    type: shapeValues[PRESET.OSC_A.shape],
  },
})
```

Figure 60. PolySynth initialization before update.



```
// Initialize PolySynth Object using Tone.FMSynth
const SYNTH_A = new Tone.PolySynth(Tone.FMSynth)
// Set PolySynth parameters
SYNTH_A.set({
  // Oscillator parameters
  oscillator: {
    // Oscillator shape
    type: shapeValues[PRESET.OSC_A.shape],
    // Unison voice count
    count: PRESET.OSC_A.count,
    // Unison voice spread
    spread: PRESET.OSC_A.spread,
  },
  // FM ratio
  harmonicity: PRESET.OSC_A.harmonicity,
  // FM depth
  modulationIndex: PRESET.OSC_A.modulationIndex,
  // Modulator parameters
  modulation: {
    // FM shape
    type: shapeValues[PRESET.OSC_A.modulationShape],
  },
})
```

Figure 61. PolySynth initialization after update.

Before the updated functionality was fully implemented, it was realised that oscillator C, which had been tuned to act as a specialised bass generating oscillator, could be updated using the same principles as discovered with FMSynth objects. Rather than use FMSynths as the voice generators however, AMSynths could be used instead. This would provide unique characteristics to the oscillator that was otherwise only limited to lower octaves, making it more appealing as an extra layer of sound to add to configurations.

Amplitude Modulation (AM) is another synthesis technique which is quite like FM. The main difference is that instead of the frequency of the carrier oscillator being modulated, the amplitude of the carrier oscillator is modulated. This results in more subtle alterations to the original waveform, which cause varying degrees of movement to the sound, which can be especially interesting when used with lower frequencies such as those in the bass and sub-bass ranges.

The only differences in the initialization of Tone's PolySynth object for oscillator C were that an AMSynth would be used as its voice generators instead of an FMSynth, and no modulationIndex property was available, as the harmonicity value would be affecting the depth of the modulation in this case.

After these changes had been made, each of the other PolySynth setter methods in the JavaScript were updated to include the new parameters, and the HTML was updated to include the five new controls for oscillators A & B, and four new controls for oscillator C. This required a few changes to the flexbox structure, as there wasn't much room for the new controls at first.



Figure 62. MS24's interface before the new controls had been added.

Event listeners were then tied to the new controls so that they would set the new parameters when changed. The new FM, AM, and unison controls were now fully functional, hugely expanding the range of sounds that could be generated by MS24.

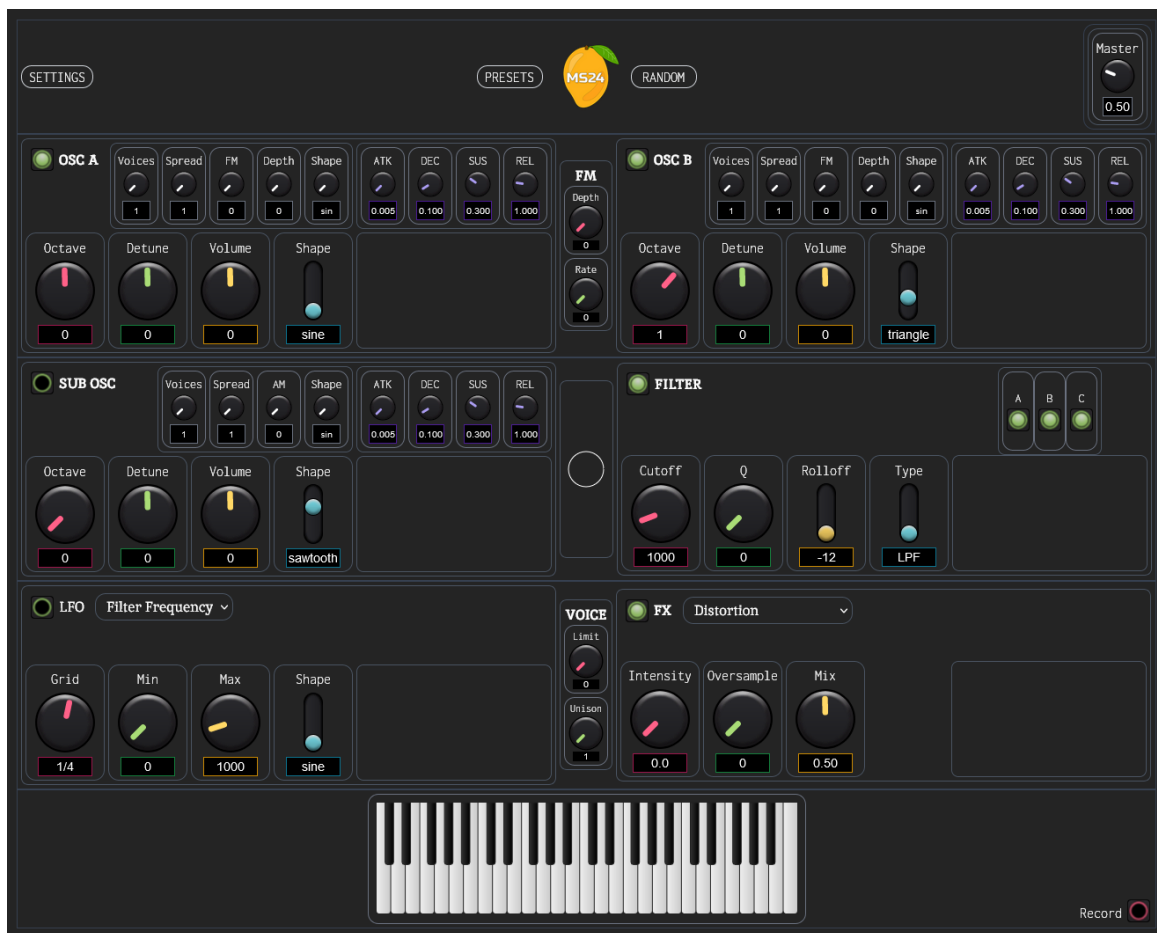


Figure 63. MS24's interface after the new controls had been added.

The overall size of the synthesizer was now taller than before, resulting in some of the synth being hidden unless the viewport was scrolled down or zoomed out. For the time being, this would be marked down as something which needed to be corrected in a future Sprint.

5.8.2 Item 2: Oscillator Visualisations: Oscilloscopes

Incorporating visualisers for the three oscillators (known as oscilloscopes) required figuring out how to "draw" on the page. Early attempts at the beginning of fourth year revealed the possibility of creating an oscilloscope by using P5.js, a library that allows writing Processing code in JavaScript. However, at the time, a single oscillator was being used instead of three, and the P5 canvas spanned the entire webpage instead of being confined to multiple small spaces.

To start with, there were two options: either creating an instance of P5 for each of the six planned visualisers or creating one instance of P5 spanning the entire synthesizer and only drawing the waveforms within the designated areas. The latter was chosen as it would be less resource-intensive and would likely result in less repeated code.

To implement this, a canvas element was created within the highest-level synthesizer div, and the z-index property was used to position it in front of the background but behind the synthesizer's controls. This allowed the controls to remain visible, and the background to be drawn on top of.

Absolute positioning was necessary for the canvas to work, as relative positioning would cause the canvas to push the other elements out of the way. The z-index attribute was used to set the element's position on the z-axis, with -10 being chosen to ensure that it was below the other elements, as all elements were initially set to z-index 0.

To test whether the canvas was working, it was necessary to draw something on it since it is an invisible element by default. To achieve this, the P5.js library was imported from a CDN, and all the code from main.js was moved to the bottom of synth.js. This was done to ensure that the P5 canvas did not start running until the synth was loaded since if the P5 logic was placed in main.js, it would not have access to the data in synth.js required to draw the waveforms.

```


// -- SCREEN SETUP -- //

let consentContainer = document.getElementById("consentContainer");
let consentButton = document.getElementById("contextConsentButton");

consentButton.addEventListener("click", function () {
  consentContainer.style.display = "none";
  synthContainer.style.display = "flex";
  if (Tone.context.state !== "running") {
    Tone.context.resume();
  }
  new p5(p5_sketch, "p5_canvas");
})

```

Figure 64. The consent button logic, now with a P5 instantiation

To accomplish this, an event listener was tied to the consent button on the welcome screen, which would hide the welcome screen and show the main synth container once pressed. An instance of P5 would then be started, linked to the element with an ID of "p5_canvas" (Figure 64).

The next step was to create the canvas and colour it to observe how it resized with the parent container. Initially, a canvas was created with the same size as the parent container, which was found by using the browser inspector. However, upon zooming in and out on the webpage, the canvas remained the same size. To address this issue, the inbuilt P5 function "windowResized()" was used, which is called every time the window size changes. Additionally, the exact size of the parent container was programmatically determined to resize the canvas accordingly.



```
let synthContainer = document.getElementById("synthContainer");
let p5_canvas = document.getElementById("p5_canvas");
let canvasWidth, canvasHeight

function getSize(element) {
  let rect = element.getBoundingClientRect();
  let width = rect.width;
  let height = rect.height;
  return { width, height };
}
```

Figure 65. `getSize()` function to return exact size of a HTML element.



```
p.setup = function () {
  // Find the width & height of the parent container
  canvasWidth = getSize(synthContainer).width;
  canvasHeight = getSize(synthContainer).height;
  // Create P5 canvas matching this width & height
  p.createCanvas(canvasWidth, canvasHeight);
}
p.windowResized = function () {
  // If the window is resized...
  // Find the width & height again
  canvasWidth = getSize(synthContainer).width;
  canvasHeight = getSize(synthContainer).height;
  // Resize the P5 canvas to match
  p.resizeCanvas(canvasWidth, canvasHeight);
}
```

Figure 66. P5 functions to setup and resize the canvas based on parent container.

Initially, `element.clientWidth` and `element.clientHeight` were used to find the size of the parent container, but upon resizing the canvas, it would end up slightly off. The oscilloscopes would be incorrectly positioned on the screen, but the reason for this was unknown. It was thought that this was due to `clientWidth` and `clientHeight` returning integer pixel values, as at different zoom levels, the inspector showed that the parent size was made up of float values.

An attempt was made to return the exact size of the element using `element.getBoundingClientRect()` and returning the width and height of this bounding box (Figure 65), but sadly, the same problem occurred.

It was decided to leave this problem for now, as it was assumed that most people would use the app at its normal size, and the issue was only noticeable due to developing with the screen split between the browser and IDE, with the browser viewport zoomed out. Later in the Sprint, it was realised that perhaps this was happening because the browser zoom resized the entire window, which has a different width/height ratio to the parent container. The possibility of spanning the P5 canvas across the entire window instead of the `synthContainer` was considered but wasn't deemed a priority for now.

The next step involved visualizing the waveforms of the three oscillators, which had been previously figured out through a YouTube tutorial in September of 2022. However, it was uncertain whether it would work with the current setup, which featured a dynamically sized canvas and multiple oscillators. Fortunately, it didn't take too long to implement the code from a previous testing project. The main challenge was positioning the visualizers, as the original code had the oscilloscope spanning the entire window.

```

p.drawWaveform = function(wave, w, h, x, y) {
  // Adjust x/y position based on current canvas size
  x = x / synthContainer.offsetWidth * p.width;
  y = y / synthContainer.offsetHeight * p.height;

  // Waveform buffer
  let buffer = wave.getValue(0);

  // Initialise start variable
  let start;

  // Find the first zero crossing
  for (let i = 1; i < buffer.length; i++){
    // if the previous point is negative, and the current point is positive/zero
    // then we've found the zero crossing
    if (buffer[i-1] < 0 && buffer[i] >= 0) {
      // Set the start to the zero crossing
      start = i;
      break
    }
  }

  // Drawing a portion of the waveform, to avoid the initial transient
  // Otherwise we would see the waveform "jump" horizontally to the start of the buffer
  let end = start + buffer.length/2;

  // Set stroke colour to white
  p.stroke(255);

  // Map x/y values and draw waveform
  for (let i = start; i < end; i++){
    let x1 = p.map(i-1, start, end, 0, w)
    let y1 = p.map(buffer[i-1], -1, 1, 0, h)
    let x2 = p.map(i, start, end, 0, w)
    let y2 = p.map(buffer[i], -1, 1, 0, h)
    p.line(x1+x, y1+y, x2+x, y2+y);
  }
}

```

Figure 67. The `drawWaveform` function, for creating oscilloscope visualisations.

In the code shown above (Figure 67), the oscilloscopes were drawn on the screen using a P5 function called `drawWaveform`, which takes inputs such as the waveform to be visualized, the width and

height of the visualisation, and its x/y position. At first, the logic didn't have an x/y input, but by examining the for loop at the bottom, it was figured out how the oscilloscope could be moved around the screen by adding the corresponding values to the four positions given to the line object.

As a result, three oscilloscopes were created that corresponded smoothly to each oscillator. Any change in the parameters while playing would immediately update the visualisations, and it was possible to achieve some fascinating shapes by playing with the frequency modulation or unison settings. This would likely be a great feature for beginner synthesizer users as it would help them to see how a waveform's shape corresponds to sound it generates.

5.8.3 Item 3: LocalStorage Presets List

Up until this point, the implementation of presets logic had been left in a basic form with two options in the dropdown, namely save and load. When selecting save, the current configuration would be saved on the user's computer, and selecting load would load a configuration and adjust the synth parameters accordingly. The main purpose of having the presets logic, however, was to provide a list of default presets to ship with the synthesizer, which would help beginners and showcase the synthesizer's wide range of sounds.

The first step was to update the HTML structure by adding a new container that would be hidden by default. This container would be placed in the main synth body, and by clicking the presets button, the synth body would be replaced with a list of presets. A placeholder list was created with buttons for saving and deleting each item. Then, JavaScript code was written to automatically populate the list with presets, followed by adding or removing presets logic.

Next, JavaScript variables were created for each of the newly added HTML elements by using `document.getElementById`. This enabled the tying of event listeners to the elements to trigger various functions when buttons are clicked.

A function was then created to toggle the presets page. When the page is closed, the "hidden" class is added to the `synthBody` element and removed from the `presetsContainer` element. When the page is open, this is reversed. This allows the placeholder presets page to be opened and closed by clicking the presets button in the top bar.

After this, a function called `populatePresetsTable()` was created to automatically add default presets to the presets page. First, the function would retrieve the presets table body using `getElementById`. Then, it would loop through the browser's `localStorage` to find keys starting with "preset-". If a match was found, the function would parse the value of this key using `JSON.parse` and add the metadata of the preset to the body of the presets table. The key would also be used as the ID for the load and delete buttons, which would have event listeners attached to them. This would allow the correct preset to be loaded or deleted from the `localStorage` based on the button clicked.

Another function called `loadDefaultPresets` was created to load 10 default presets for testing from the newly created presets folder. This function required an array of preset names to loop through and make HTTP GET requests to load them on both local and remote versions of the application. The response from these requests is then parsed as objects and added to the browser's `localStorage` using `localStorage.setItem`.

To prevent duplication of entries on every page load, a function called `checkForPresets` was created to loop through `localStorage` and check for keys including "preset-". If no presets were found, it

would then call `loadDefaultPresets` to populate the list with default presets. This ensures that the presets list is always populated with default presets for new users and will only be reloaded if all the presets in the list are deleted.

Following this, some of the existing logic was made into functions, so that they could be reused with the new `localStorage` preset logic. The first of these functions was `loadPreset`. Originally the logic was only being used to load from disk, but now there were two possible origins of a preset, so it made the most sense to reuse the existing code rather than have two large and almost identical blocks of code. The load logic was also updated to include the new preset metadata, so that the currently loaded preset could be easily displayed. The logic used to save a preset to the user's computer was then moved into a new function named `downloadPreset`, before creating `savePreset` for saving to the `localStorage` and `deletePreset` for removing them.

Lastly, event listeners were added to the `presetContainer` buttons, and the corresponding functions were assigned to them. Additionally, the input elements were also assigned with event listeners for the four metadata properties that were required when saving a new preset. The save function was updated to alert the user if they didn't provide a name, while the other fields were left optional. The download function was also updated to use the metadata name as the filename instead of a generic name.

With this logic completed, the presets implementation was a lot closer to that of a traditional software synthesizer. The new functionality would improve the usability of MS24 while also making it more beginner friendly.

5.8.4 Item 4: Lossless Recording

As discovered in the previous Sprint, lossless recording was not possible through Web Audio's `MediaRecorder` API. During Sprint 6, three different libraries were experimented with to expand the capabilities of Web Audio's `MediaRecorder` API by offering extra encoding options. One of the tested libraries, named `libflacjs`, brings the FLAC codec to the `MediaRecorder`. FLAC is the most preferable lossless format as it maintains audio fidelity that is identical to the source material while also compressing the data to make file sizes smaller than WAV files, which are uncompressed. WAV files treat every second of audio in a file uniformly, irrespective of amplitude information. As an example, if silence for one minute is recorded and then encoded as WAV, and white noise for another minute is also encoded as WAV, both files would be of identical size. WAV files were initially intended for very short sound effects, but over time became the standard for all lossless audio, resulting in very large files when it comes to music. On the other hand, FLAC leverages compression algorithms that are similar to those used in compressed archives, such as ZIP files. Hence, a silent file would only be a few kilobytes, with only the metadata and encoded information occupying storage space.

The issue encountered with `libflacjs` was the lack of clear documentation on how to use it with Vanilla JavaScript. While there were guides available for using it with Node.js and JavaScript frameworks like React, there were no dedicated instructions for Vanilla JavaScript. Despite attempting to import it anyway, all efforts to make it work were unsuccessful. Other libraries were attempted, and if no viable solution was found, the plan was to send a blob from the client side to the server, where it would be encoded as FLAC and returned to the client for download.

The next library attempted was called opus-recorder, but much the same as libflacjs, this only provided information on how to get it set up in Node.js or with a framework using WebPack, a build tool like Vite.

In the end, extendable-media-recorder was discovered as a solution by using a slightly different search prompt. This library claims to be an "extendable drop-in replacement for the native MediaRecorder," allowing the definition of custom encoders to render files unsupported by any browser. After installation through node package manager, along with the custom encoder extendable-media-recorder-wav-encoder, the library was imported without much difficulty. However, when changes were pushed to GitHub, an issue arose due to the asynchronous method of importing the custom encoder, which was unsupported by Vite to protect against older browsers that cannot compile asynchronous code at the top-level of a file. To address this, a Vite plugin called vite-plugin-top-level-await was discovered, which converts any top-level JavaScript awaits into a syntax compatible with older browsers.

By using this library, it would now be possible to stop using Tone's inbuilt Recorder object, limited by the native MediaRecorder, and instead rewrite the Tone Recorder's logic with the extendable MediaRecorder.



```
const REC_DEST = Tone.context.createMediaStreamDestination()  
const REC = new MediaRecorder(REC_DEST.stream, { mimeType: 'audio/wav' });  
let CHUNKS = [];
```

Figure 68. The three new variables required for lossless recording.

To do this however, three new variables needed to be created: REC_DEST, REC, and CHUNKS.

As can be seen in Figure 68, the REC_DEST variable was created using a MediaStreamDestination node from Tone's Audio Context. This node is responsible for creating an audio stream of bytes from a specified source. This audio stream is then used as the input in REC, a MediaRecorder object using the extendable-media-recorder version of the object rather than the native one. The mimeType is also set here and is set to the lossless 'audio/wav' instead of the native version's default: 'audio/ogg'. Finally, CHUNKS is initialised as an empty array, so that later it can be filled with audio chunks before creating a lossless blob to download.

For these new variables to allow recording, the synth's master output node had to be connected to the REC_DEST node, so that any recordings started would use the output of the synthesizer as its source.

Next, the recording toggle button logic was updated so that it started the new MediaRecorder instead of Tone's Recorder when toggled on and stopped it when toggled off. On top of this, the logic was updated so that when toggled on, the CHUNKS array would be reset, as otherwise it would keep the old recordings in memory.

After this, two MediaRecorder methods were used to handle the data, which was a little different to Tone's Recorder logic.

The first of these methods was `ondataavailable`, which, as it sounds, fires once data is available to the `MediaRecorder`. There was only one line required here: `“CHUNKS.push(e.data)”`. This line ensured that every bit of the audio stream would be added to the `CHUNKS` array while the recorder was recording.

```

// once data is available to the recorder...
REC.ondataavailable = (e) => {
  // log to console
  console.log('DATA AVAILABLE!', e.data)
  // push data to chunks array
  CHUNKS.push(e.data)
}
```

Figure 69. `MediaRecorder`'s `“.ondataavailable()”` logic

The other method was `.onstop`, which gets fired once the `MediaRecorder` is stopped.

```

REC.onstop = (e) => {
  // create blob from chunks
  let blob = new Blob(CHUNKS, { type: 'audio/wav' })
  // create download link
  recordingURL = URL.createObjectURL(blob)
  // create anchor element
  const anchor = document.createElement('a')
  // set file name & format
  anchor.download = 'recording.wav'
  // set anchor href to url
  anchor.href = recordingURL
  // click anchor (download)
  anchor.click()
  // return label to default
  recorderLabel.innerHTML = 'Record'
}
```

Figure 70. `MediaRecorder`'s `“.onstop()”` logic

The code shown in Figure 70 creates a blob from the chunks array with the 'audio/wav' type. Following this, it creates a URL object using the newly created blob. After that, it creates an anchor element in the HTML and sets the "download" property to specify the filename and extension of the downloaded file, and the "href" property to assign the URL of the anchor to the URL object created earlier. Finally, the .click() method is called on the anchor element to initiate the download of the encoded WAV file to the user's computer.

To check if this was truly a lossless WAV file, and not a lossy file with the wrong extension, two desktop applications were used. The first of these applications is called MediaInfo, which displays all the metadata information contained in media files opened with it, including container information, and encoding information. This had also been used during the first implementation of the recorder to see what file extension should be given to the lossy files, as it also informs of whether a file's extension is valid or not.

In the case of the implementation of the lossless recorder, MediaInfo was used to check the real format and file extension of the rendered WAV file. The results showed that the extension was in fact correct, and the audio format was PCM, which is the expected format for a WAV file.

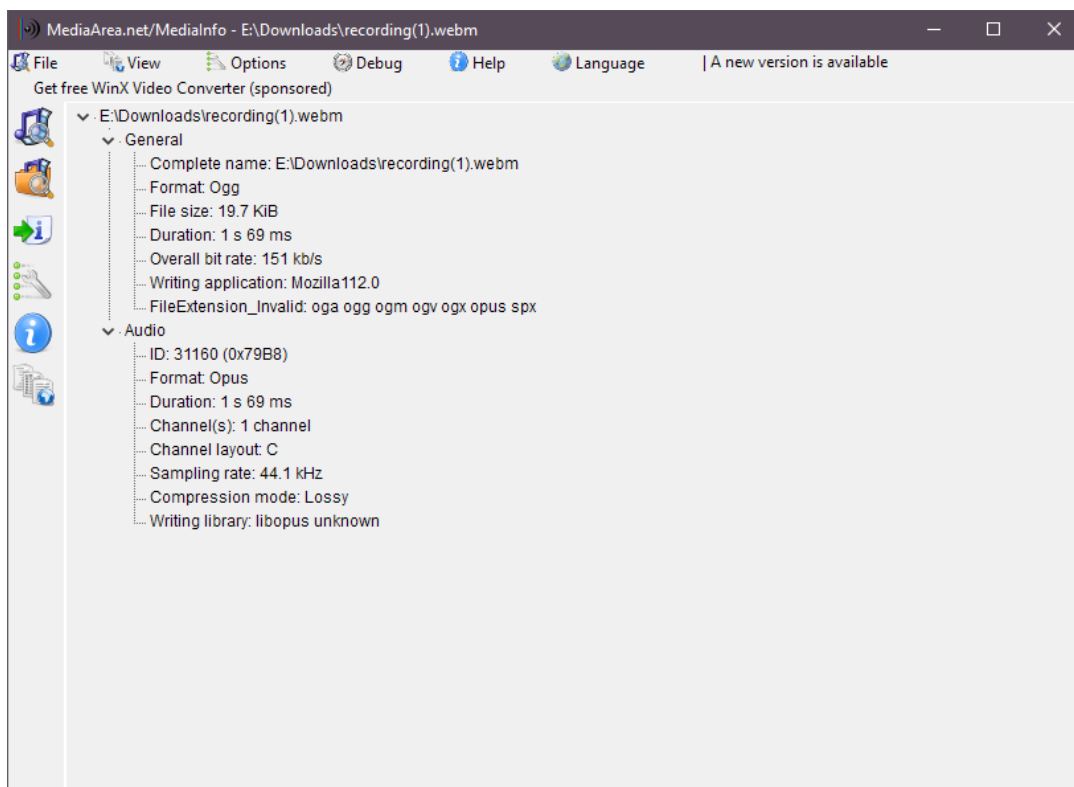


Figure 71. MediaInfo reading a .OGG file incorrectly given a .WEBM extension.

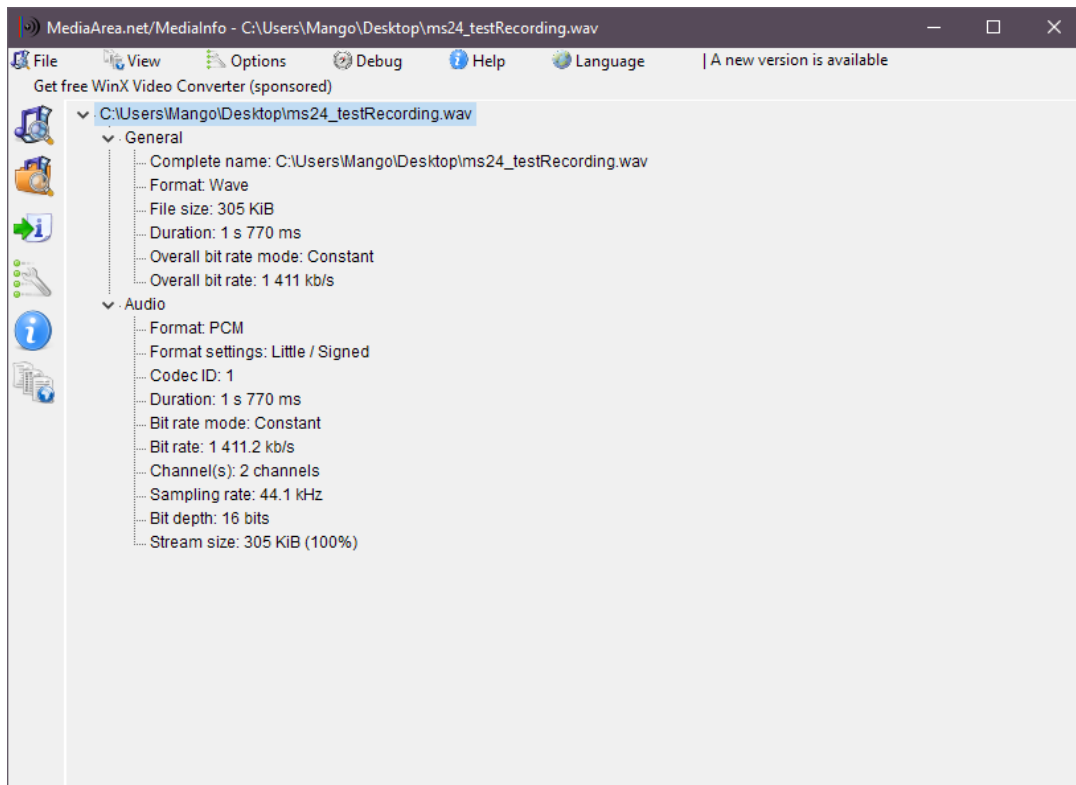


Figure 72. MediaInfo reading a legitimate .WAV file

Next up came Spek, a helpful tool for manually verifying whether an audio file is truly lossless or not. It provides a spectrograph of the audio file, which allows you to visually inspect its data. Although MediaInfo had already confirmed that the encoding of the newly created WAV file was lossless, it was still necessary to verify whether Tone.js running in a browser was outputting lossless audio or not, as it was still possible that the output was lossy, but there was no way of knowing for sure without a lossless recording.

By loading a loud recording into Spek it was observed that the frequency data stretched all the way up to the ceiling of the spectrogram, which is 22kHz for 16-bit, 44,100kHz audio. This confirmed that the audio output of Tone.js and the browser was indeed lossless, as if it were lossy, there would be an easily identified line cutting the top off, usually around 15-20kHz, where it is safest to remove and compress data without perceivable quality loss.

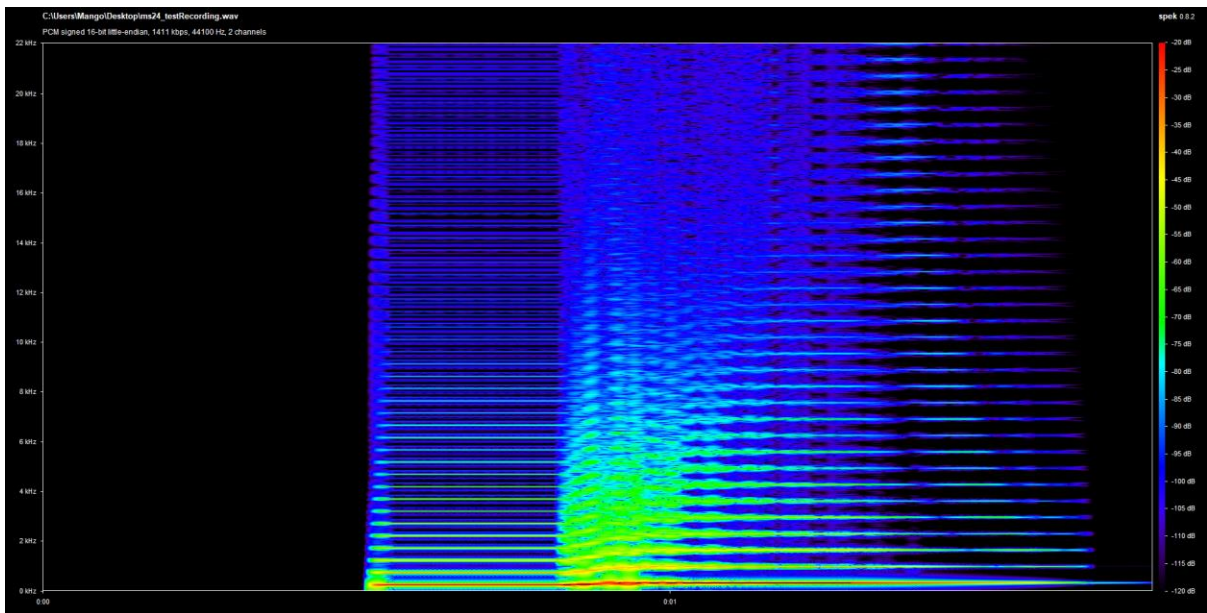


Figure 73. Spek reading a lossless WAV recording, output from MS24.

After verifying that the output was lossless audio, these changes were pushed to GitHub, which in turn updated the hosted version on Vercel. When opening the hosted version in Firefox and testing the recording a few times, it was noticed that it randomly failed to record. It was then realised that Chrome had been used almost exclusively while implementing this change, with no testing done in other browsers. Chrome has been the browser of choice throughout development since the V8 engine is a lot faster at compiling JavaScript than Firefox's SpiderMonkey engine, which can sometimes struggle under high load. Upon further investigation, it seemed that it was only Firefox that would break this way, and weirdly, only occasionally.

To encounter the bug, you would have to start a recording, play some keys, stop the recording, then repeat. Eventually, upon toggling the recording switch off, an error would be logged to the console, and nothing would be downloaded. After lots of troubleshooting and hours of exploring the debugger, no reason for this could be found. It was assumed that either Firefox was incapable of lossless recording, or there was an issue within the library's source code.

To investigate a potential solution, the mimeType was reverted to 'audio/ogg' and several recordings were made in Firefox. Surprisingly, no matter how many recordings were started and stopped, it would export the recording without any issue. So, to bypass this issue and still allow Chrome (and other V8-based browsers) to record in lossless, a library called browser-detector was installed. This library simplifies the process of making the current browser and other userAgent information available within JavaScript and allowed the addition of an if-statement which changed the mimeType depending on which browser was being used. This worked a charm, but it was a bit disappointing that Firefox was seemingly unable to record in lossless, as the goal was to achieve optimal cross-browser compatibility.

In case the cause of the issue was not Firefox, [an issue](#) was created on GitHub in the extendable-media-recorder repository with detailed context. The issue was explained clearly, including the attempted solutions, and code blocks were used with Markdown syntax to enclose specific lines and the complete error message from the console. Additionally, links to the exact line where the error occurred on two different commits were provided.

Two days later, a response was received from the author of extendable-media-recorder, identified as @chrisguttandin on GitHub. The author expressed gratitude for the issue submission and described the situation as "definitely an interesting one." The author also stated that the extendable MediaRecorder logic depends on a library called standardized-audio-context, which is also used by Tone.js. Additionally, the author assured that no global objects were patched to enable multiple instances or versions of standardized-audio-context to be used concurrently. The author mentioned a possibility of a race condition in the [WebAudioMediaRecorder](#), the internal recorder utilizing the Web Audio API, specifically in Firefox. It was also inquired if the 'stop' event is always waited for before triggering another 'start' event. A reply was sent back to the author after over a day, stating that the code had been made more robust by checking the state of the MediaRecorder to ensure it was inactive before starting another recording, and to ensure it was recording before stopping. Despite this, the issue was not resolved, and a reply was sent back explaining that the issue persisted.

The author replied shortly after, suggesting that the toggle logic should be sufficient and explaining that the issue might be with his library. A link to the source of the error message was shared, and it was revealed that the AudioWorkletProcessor responsible for audio recording was being set to an undefined state for unknown reasons. While examining his code, he observed that there was a possibility of the recorder stopping itself when it received only silence. As a result, there was "no way to communicate that back to the code which initiated the recording, and consequently it sometimes tried to stop the already stopped recording", causing the error to occur. He concluded by expressing gratitude for the detailed bug report, which was instrumental in resolving the issue.

Following this, the version of extendable-media-recorder in the application's package.json file was updated from 7.1.9 to 7.1.10, and the logic that set the MediaRecorder's mimeType to 'audio/ogg' on Firefox was removed. After extensive testing with many recordings, the error did not appear, whereas it would usually occur every three or four recordings before.

This enabled lossless recording to be available on all browsers. It is a good example of why open-source code communities such as GitHub are valuable. The bug might not have been discovered if the issue had not been submitted, and it may have taken a long time to resolve without this collaboration.

5.8.5 Item 5: Electron Experiment

A meeting with the project supervisor took place during Sprint 6 in which a framework called Electron was discussed, which is used by many modern web-based applications as the foundation for a desktop version of that application. Many popular applications, such as Microsoft Teams, Visual Studio Code, GitHub Desktop, and Discord use Electron so that one codebase can be maintained while providing for both web versions and desktop versions of the application, without the need for any specialised native code.

It does this by essentially packaging a Chromium browser engine with the Node.js runtime environment, stripping the browser of the tell-tale signs such as the toolbar, address bar, and bookmarks bar, leaving only the viewport visible, while disabling default behaviours like right-click context menus and keyboard shortcuts. Taking a web application as its input, once built, the output is several binary executables, for each of the most common operating systems.

After having noticed that MS24's performance varied between browsers, especially between browsers using different JavaScript engines, such as Firefox's SpiderMonkey engine versus Chrome's V8 engine, it was decided that trying out Electron would be a good idea, since this would provide a standalone version of the application using the V8 engine, which in theory could provide better performance than any other method, since the browser used by Electron binaries is completely stripped down, lacking extensions and other variables that could impact the application's performance.

To get up and running, Electron's [quick start guide](#) was used, which takes you through the process of creating a barebones app in Electron. This process was surprisingly simple, consisting of just three files: `main.js`, responsible for setting up the Electron instance, `preload.js`, responsible for preloading global variables such as `window` and `document`, and `index.html`, used as an example of the source code to be packaged.

To see if the code worked, the `start` command located in `package.json` was executed, telling Electron to bundle the app and locally host it. Rather than returning a URL to be opened in the browser like the `start` script does with JavaScript frameworks and build tools such as Vite or React, Electron opened a new window, dedicated to the application, showing the contents of `index.html` within an 800x600px window, as per the default settings from `main.js`. The `make` command was then tested, which creates distributable binaries (executables) for the current operating system. This command output a folder called "out" containing one directory with the distributable and one with the packaged application code. By running the executables, the example window opened again without issue. The idea is that these files get included in the release of an application, allowing anyone to install the web app on their computer, allowing for offline access.

Following these successful tests, some further research was done to see how to go about packaging a pre-existing web application such as MS24, as it wasn't clear whether to use the pre-built source code or the built files from Vite. Eventually it was discovered that the built files should be used, with slight manipulations required for the import URLs from `index.html` to work with Electron.

The MS24 project was returned to, and the `build` command was executed in the terminal, packaging the web application in a folder called "dist". This folder was then copied to the new `ms24_electron` directory, so that it could be used instead of the example `index.html` file.

Returning to the Electron project, `main.js` was updated so that instead of loading `index.html` from the root directory it would instead load the `index.html` from the `dist` directory. The width and height of the window were then set to 1920x1080px, and the `start` command was executed. While this resulted in a window opening, it was bare HTML with no styling or functionality. Upon examination of the inspector, the assets minified by Vite were failing to load. This was because the format of the import URLs inside `index.html` hadn't been corrected to exclude the leading forward slash. For example, what was `"/assets/logo.png"` would have to be `"assets/logo.png"` before Electron would package them correctly.

Surprisingly this was all that was required to implement an Electron version of MS24. Upon executing the `make` command, it was able to output MS24 binaries which could be distributed and installed on any 64-bit Windows operating system. A few minor changes were made after this, such as setting the icon for the tab, hiding the title bar, and setting the colours of the minimize, maximize, and close buttons on the top-right of the window.

Electron would not be included in the main MS24 repository, as this would disrupt Vercel hosting without a confusing restructuring of the directories. Mixing and matching the package.json files didn't seem like a good idea, and nesting the electron directory wasn't likely to work either.

For now, the Electron directory would be left separate. When it comes to the official release of MS24, it will be used again to build the final version of the application, so that the binaries can be included in the main repo and release.

5.8.6 Item 6: User Testing

For the user testing of MS24, it was originally planned to use the online tool Loop11 to create and conduct tests, as it allows for remote moderated and unmoderated testing by overlaying a web application with tasks and questions, with several powerful functionalities provided such as URL listening and optional webcam and microphone recording. However, after making a test consisting of five tasks and eight questions using the same student account as used before for other projects, it was discovered that the educational license had expired. Since time was of the essence, it was decided that the tests would be conducted manually in-person with friends and family, using the same tasks and questions as made with Loop11.

The testing was conducted in a controlled environment to ensure consistency and accuracy of the results. Participants were given a brief overview of the project and what was expected of them during the testing. They were then asked to complete a set of tasks, such as creating a basic sound and modifying, and recording the output of the synth through the inbuilt recorder. During the testing, participants were encouraged to speak aloud and provide their thoughts on the interface, the synthesizer's functionality, and any issues they encountered. After each of the tasks had been completed, participants were asked to answer eight questions about the ease of each task and overall feedback on the application and its GUI. All but one of these questions were in the Likert scale format, with five options: strongly agree, agree, neutral, disagree, and strongly disagree. This was done to provide quantitative value to otherwise qualitative data.

After the testing was complete, the results were analysed to identify any patterns or common issues that arose during the testing. Any usability issues were marked down, so that improvements could be made to the user interface during the next Sprint, based on the feedback received.

The results of the user testing process can be found in the next chapter.

5.9 Sprint 7

The goal for Sprint 7 was to add some finishing touches to MS24. This would consist of fixing major bugs, as well as adding missing functionality, such as arpeggiation and theming. It was also important to make the application more user-friendly, by adding tooltips to explain each part of the synth.

5.9.1 Item 1: Keyboard Logic Rework

The initial implementation of keyboard input logic was far from ideal. There were many bugs and limitations, primarily due to using the **webaudio-controls** GUI keyboard as the main event handler. For this reason, the keyboard logic needed to be reworked.

The problem with using the library's GUI keyboard was that for events to be fired by the element, it had to be focused. Focusing on a browser is primarily done with the mouse; anything you click on will get focus, removing focus from the previous element. This meant that any internal event handling done by the library, such as keyboard and mouse event listeners, would no longer function if anything but the GUI keyboard itself was focused on. When using virtual analogue synthesizers, it's very common to hold keys while adjusting parameters, so that changes can be heard immediately. However, with the logic at this point, doing this would cause the key (or keys) held down to "stick" down, so that even if the physical keys on the computer keyboard were released, the keys on the GUI keyboard would remain held, since without focus, it was unable to fire the necessary "keyup" event used to trigger a release event for the notes being played.

Another trigger for the "sticky key" issue was changing an oscillator's octave or semitone offset while holding keys. While this was possible to do without clicking outside of the GUI keyboard, by using the mouse wheel, it had the same result, as the logic didn't keep track of which keys were being played versus which were being held. If when holding a C3 note on oscillator A, you changed oscillator A's octave offset to +1, the desired behaviour would be that the C3 would switch to a C4, but instead C3 would remain held indefinitely, regardless of whether the physical key had been released. This is because the release logic would try to release a C4 when the synth was playing a C3.

The last reason for reworking the keyboard logic was that using the library's GUI keyboard meant being limited to the functionality provided by it. While the library does provide support for computer keyboard and mouse inputs, as well as each of its controls being MIDI-learnable, it seemed that controlling the GUI keyboard with a MIDI keyboard wasn't possible.

Due to this, the keyboard logic rework started with MIDI support. First and foremost, this required creating a class called "MIDIAccess" which defined a MIDI access object that could be used to interact with any connected MIDI device. The class was given five methods; start, initialize, initializeDevice, onMessage, and requestAccess.

5.9.1.1 *MIDIAccess Class*

The start method of the class returns a Promise which resolves if access to MIDI input devices is granted and rejects if there is an error. It calls the initialize method of the class, passing in the 'access' object, which initializes each input device by calling the initializeDevice method on each device.

The initializeDevice method sets the 'onmidimessage' property of the device to 'this.onMessage.bind(this)', which sets up a callback for when a MIDI message is received from the device. It also logs the name, state, and type of the device.

The onMessage method is the callback, which is called when a MIDI message is received. It extracts the command, note, and velocity values from the message and calls 'this.onDeviceInput' with an object containing these values.

The `onDeviceInput` function is defined outside of the `MIDIAccess` class. This function extracts the command, note, and velocity values from the message, logs them, and then calls the function `handleNote`, passing in the same values.

5.9.1.2 *handleNote Function*

This function takes in the state, note, and velocity of a MIDI message and performs actions based on the state. If the state is "on", it calculates the frequencies of the note for each of the three oscillators and triggers the corresponding attack for each oscillator that is enabled. If the state is "off", it triggers the corresponding release for each oscillator and removes the original frequency from the list of playing frequencies.

There two helper functions used by `handleNote` to determine the correct frequency to trigger for each oscillator. The first of these is called 'midiToFreq', which converts a MIDI note number to a frequency. The second is called 'frequencyOffset', which calculates the offset for a given octave and semitone. Combining these helper functions lets us find the exact frequency to trigger for each oscillator when a key is held down, taking the octave and semitone offsets of each oscillator into account.

Finally, there are two global arrays used by `handleNote`: 'playingFrequencies' and 'synthPlaying'. These are used keep track of the currently playing frequencies and whether the synthesizer is currently playing any notes. Keeping the currently playing frequencies in state would be essential for fixing the bugs presented by changing an oscillator's octave or semitone offsets while playing notes.

5.9.1.3 *changeNote Function*

To correctly handle note offsets being updated while playing, a new function called 'changeNote' was created. This function takes three notes or frequencies as an input, and then checks to see if the synthesizer is playing any notes, by referencing the global `synthPlaying` state variable. If it is, the function fires a release signal to each of the oscillators using the input notes, adjusts the notes based on the current octave and semitone offset values, and then fires an attack signal to each of the oscillators with the updated frequencies. `changeNote` was then added to each octave and semitone offset event listener, using the global `playingFrequencies` array as the input.

Returning to the earlier example with this updated logic, if one were to hold C3 and adjusted the octave offset to +1, `changeNote` would release the C3 note, adjust the note with the `frequencyOffset` function, and then trigger an attack using C4 as the new note.

5.9.1.4 *guiKeyActivator Function*

Even though MIDI keyboard support had now been implemented, holding MIDI keys wouldn't update the GUI keyboard like the old logic did. This is because the old logic used the GUI keyboard's internal event listening which also controlled which keys lit up. MIDI note handling was entirely separate to the internal event listening, so a new function had to be created to mimic this behaviour. This function, "guiKeyActivator", takes two inputs, a note, and the state of that note (on or off), before using simple maths to find the correct value for the GUI keyboard. Luckily, the `webaudio-controls` library included a method called `setNote`, which could be used to manually set the state of

the keyboard. The state and calculated key get passed to this setNote method, so that the GUI keyboard's keys light up when the corresponding key is held on the MIDI keyboard.

5.9.1.5 *Migration of Old Logic*

To fix the sticky note bugs, rather than modifying the old logic which still controlled the mouse and computer keyboard events, an experiment was made to see if the new logic could also handle these input types.

A "keydown" and "keyup" event listener was added to the document, so that any keys pressed on the computer keyboard would emit an event. A new function was created to handle these events, namely "handleKeyEvent". This function takes an event as an input, which it then checks to determine what state the key is in. If it was given a keydown event, the state would be "on", whereas if it was given a keyup event, the state would be "off".

Following this, a switch case was created, using "event.key" as the target. In each case, the handleNote function would be called, using the state determined by the event type, followed by a key signal determined by the key being held. For example, if the key "Z" was held on the keyboard, the handleNote function would be given "on" as the state, and "0" as the note value. This would handle the Z key being held the same way as if the first key on a MIDI keyboard was held, and better yet, it worked despite the GUI keyboard's focus state.

Combined with guiKeyActivator, the correct GUI keys would light up when keys on the computer keyboard were held. This rework made the old logic almost entirely redundant. The only remaining input type left to rework was the mouse input.

After the redundant keyboard event logic had been removed, it became apparent that the GUI keyboard's events were compatible with the handleNote function. This was good, because the only simple way to handle mouse inputs properly was to use the GUI keyboard's native event handling. Otherwise, it would have required tying event listeners to each individual GUI key, before mimicking the functionality entirely.

When clicked, the keys on the GUI keyboard emit an array holding two important values. The first of these values is a Boolean representing the state of the key, meaning if the key is pressed, the value is 1, and if it is released, the value is 0. The second of these values is the key index. The first key on the GUI keyboard emits 0, the next emits 1, and so on. Due to this, it was relatively simple to plug these values into handleNote, meaning that all input methods now used the same function to handle note playback.

The handleNote function did need to be adjusted slightly to allow for this however, with "origin" being added to its inputs. This would be used to determine the type of note input that had triggered the function, so that the correct note frequencies could be calculated in each case. For example, if the origin was a MIDI keyboard, the note value would be reduced by 48 before the note was handled. This was to ensure that every input method was on the same starting octave. After these adjustments had been made, clicking the first key on the GUI keyboard, pressing the Z key on the computer keyboard, or pressing the first key on a MIDI keyboard would all play the exact same note.

The only issue left by this point was that the GUI keyboard's native event handling would interfere with the new custom event handling when the GUI keyboard was focused on. This is because for the mouse input to work, the keyboard's change event was still being listened to, and this would emit

when a key on the computer keyboard was held as well as if clicked with the mouse, and the array emitted doesn't contain any origin information.

To get around this while retaining mouse input functionality, the webaudio-controls library had to be edited directly. To do this, the library first had to be downloaded and added to the application locally, as it was being imported from a CDN (Content Delivery Network) before. Next, the library was opened in WebStorm before using the find function to search the file for any mention of "keydown". Sure enough, this search returned a bunch of keydown events, each of which were responsible for part of the library's computer keyboard event handling. After commenting-out these lines to deactivate the functionality, the application was heavily tested to ensure nothing had broken. Luckily, everything was fine, and the GUI keyboard no longer emitted keyboard events when focused.

5.9.1.6 *Global Octave Modifier*

While working on the new keyboard logic, it became apparent that there was a common synthesizer feature missing from MS24. This feature was a global octave offset. While each oscillator already had octave and semitone offset controls, when combined with the computer keyboard's two playable octaves, the synthesizer only had a maximum of seven playable octaves. This seemed a bit low, so it was decided that a new GUI slider should be added, to control a global octave offset which would adjust all oscillators at once.

This slider was given a default value of 0, a minimum of -2, and a maximum of +2, resulting in 11 possible octaves with any input device.


The handleNote and changeNote functions were then adjusted to use this global octave offset when calculating frequencies to trigger. changeNote was also added to the global octave control's event listener, so that every oscillator would seamlessly change octave if the synthesizer was currently sounding.

5.9.1.7 *Arpeggiation*

The last feature of the keyboard logic rework was the addition of arpeggiators. Arpeggiators are a common feature in synthesizers that automatically play a series of notes one-by-one in various patterns. For example, if an arpeggiator is enabled while holding a C Major on the keyboard (notes C, E, and G), rather than all three notes sounding at once, the arpeggiator will split the chord out into its individual notes, before playing through them in the specified sequence or "pattern". Common patterns include "up", "down", "up down", and "down up". The up pattern would play the notes in the following order: C, then E, then G. Whereas the down pattern would play G, then E, then C.

The rate at which an arpeggiator plays notes is usually defined in bars or subdivisions of bars and is usually synchronised to the clock of the synthesizer and/or DAW. A rate of 1/4 would mean four notes are played every bar, whereas a rate of 1/1 would mean only one note is played every bar.

When it came to implementing this feature, it was a question of figuring out Tone's Pattern class. Initially it was attempted with a single Pattern object, which would trigger each oscillator at once with the given array of notes. The problem with this was that it couldn't offset the notes played by each oscillator's octave and semitone values, and it wouldn't be possible to turn the arpeggiator on for some oscillators while off for others.



```

const ARP = new Tone.Pattern(function(time, note){
  SYNTH_A.triggerAttackRelease(note, 0.25);
  SYNTH_B.triggerAttackRelease(note, 0.25);
  SYNTH_C.triggerAttackRelease(note, 0.25);
}, ["C4", "D4", "E4", "G4", "A4"])

```

Figure 74. Initial Tone.Pattern attempt

For this to work with the offsets and to allow for individual oscillator arpeggiation, it was necessary to create a Tone.Pattern object for each of the oscillators. Each Pattern would be identical, other than their callback functions which determine which oscillator to trigger, and their note arrays, which determine which notes get played.



```

const ARP_A = new Tone.Pattern(
  // Callback function (what to do for each note)
  function (time, note) {
    // Trigger SYNTH_A attack & release
    // (note to play, duration of note, time before release)
    SYNTH_A.triggerAttackRelease(note, '16n', time)
  },
  // The array of frequencies to loop through
  SYNTH.STATE.arp_A_frequencies,
  // Arpeggiator pattern to use (up, down, upDown, etc.)
  PRESET.ARP.pattern
  // Arpeggiator rate (1 = 1/4 bar)
).set({ playbackRate: PRESET.ARP.playbackRate })

// Same for ARP_B and ARP_C...

```

Figure 75. Correct Tone.Pattern initialization

Three new state variables were created, one for each note array, so that the note offsets of each oscillator could be updated independently before the arpeggiators were triggered. Two functions were created to handle the arpeggiation, startArp and stopArp. Both functions take three inputs, frequency A, B, and C. If arpeggiation is enabled for a particular oscillator, the corresponding frequency is pushed to the corresponding note array, before the arpeggiator's setter method is called to update its values property. The arpeggiator in question is then started and will play through held keys in sequence while active.

The stopArp function does the reverse of startArp, in that it filters the input notes out of the note arrays before setting the arpeggiator values and calling the stop method.

By this point, arpeggiation was working as intended, and new sliders and switches were added to the interface to control the arpeggiators with. A toggle was added to each of the oscillators, for turning each arpeggiator on or off, two sliders were added, one for selecting the arpeggiator pattern, and one for selecting the playback rate, and one knob was added, to control the BPM (Beats Per Minute) of Tone's Transport, which each arpeggiator is synchronised with.

One issue, however, was that changing octave or semitone offsets while any of the arpeggiators were enabled would cause a bug just like the sticky keys bug from before, with the original frequencies never releasing due to the updated values. To fix this, the changeNote function had to be updated so that if an arpeggiator was enabled while offsets were updated, the original notes would be filtered out of the notes arrays before being replaced updated ones. On top of this, Tone's Transport object had to be started and stopped depending on the number of keys being held. Without this, changing offsets while playing would seem fine at first, but the next time notes were held, there would either be a severe delay before they played, or they would simply not play at all.

5.9.1.8 Double keyDown Bug

An uncommon bug that had been encountered a few times earlier in the development of MS24 resurfaced during the implementation of the arpeggiators. This strange bug didn't seem to be due to the logic of the application at all, but rather an issue with keydown and keyup event listeners in the browser.

To replicate this bug, it required holding one key on the keyboard, before quickly switching to another key and holding it down. If done quickly enough, a phantom keydown event would be triggered, as if the second key had been held twice. This bug was problematic, as certain events were dependent on the number of keys being held down at any given time, but this number was calculated by using the event listeners that caused the bug. When any keydown event was triggered in the application, the heldKeys variable would be incremented, and if any keyup event was triggered, it would be decremented. The phantom keydown bug meant that even after releasing all keys on the keyboard, as far as the application was concerned, there were still keys being held down, meaning logic dependent on this variable would not trigger.

To fix this, a new variable was added to an object containing all stateful values in the app. This new variable, lastKeyDownEvents, would be pushed to whenever a keydown event was triggered. The rest of the logic within the keydown event listener was then wrapped in an if statement which checked to see if the current event existed in the array. If it did, the logic would no longer fire, meaning the heldKeys variable could now be trusted.

This change didn't exactly "fix" the bug per-se, but the issue was resolved.

5.9.2 Item 2: GUI Rework

The second item worked on in Sprint 7 was a rework of the GUI. Ever since the implementation of FM, AM, and unison controls at the beginning of Sprint 6, the synthesizer had become too tall to fit within the browser's viewport, which was a common complaint in the results of user testing. Another GUI issue was missing visualisations. While the oscillators had been given oscilloscope

visualisations in Sprint 6, the filter, LFO, and FX groups visualisation boxes were empty, which didn't look great.

5.9.2.1 Resizing

To start with, the largest elements of the interface were made note of, so that they could be tackled one by one. The first of these elements were the primary GUI knobs and sliders, which were much too big compared to the other controls. While the small knobs were set to 30px in diameter, the primary knobs and sliders had been set to 70px. Shrinking them to 50px was an immediate improvement, as they were still larger than the secondary knobs, but a lot less out of proportion.

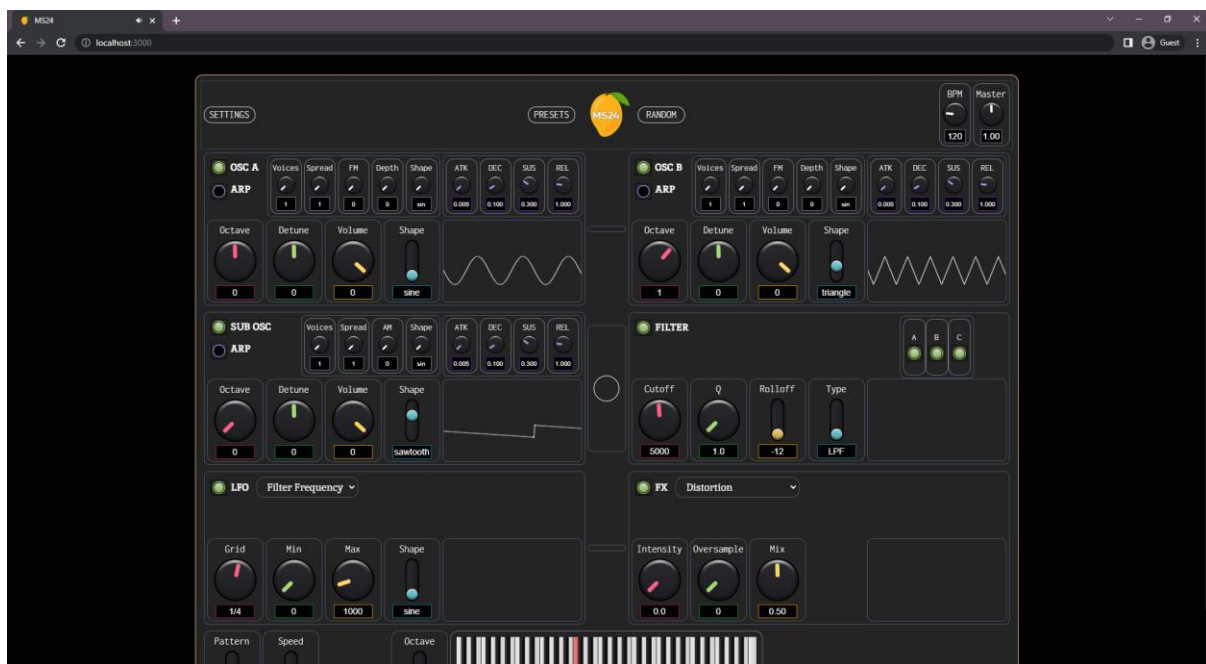


Figure 76. MS24 before the GUI rework

Changing the primary knob diameters wasn't enough to reduce the height of the synthesizer interface however, as the visualisation boxes were still set to match the height of the 70px controls. After some adjustments to the canvas elements and CSS, the synthesizer was around 60px shorter than before. It was still too big for the viewport though, so more changes needed to be made.

The GUI keyboard was the next element that would be easy to adjust. Starting at a height of 128px, it was far too tall, which was made especially clear now that the primary knobs had been made smaller. Setting the height to 85px looked a lot better, matching the 70px tall sliders beside it.

The synthesizer interface was now very close to fitting inside the viewport, but it was still a few pixels off. To fix this, the padding between elements was adjusted so that it was more uniform, as before there were some elements using a padding of 0.5rem, while most of the elements were using 0.25rem. The padding outside of the synthesizer container was also changed from 2rem to 1rem.

Lastly, it was noticed that the master gain and BPM controls on the top row of the synth were using a diameter of 40px. These were reduced to 30px to match the other small knobs within the interface.



Figure 77. MS24 after the GUI rework

After these changes had been made, the synthesizer now fit inside the browser viewport, with no scrollbar being displayed anymore. The interface now looked better too, due to the more consistent spacing and sizing across the board.

5.9.2.2 Theming

Next came theming. While basic theme switching had been implemented in Sprint 6, the light theme still left a lot to be desired. This was mostly due to the issues with importing Tailwind's configuration while using Vite and vanilla JavaScript, since without the Tailwind configuration, it seemed like it would be impossible to update Tailwind's colour variables on the fly.

Several attempts had been made to use the `setAttribute` method on each element to manually update the classes used, so that one Tailwind class could be swapped out with another (i.e., "border-gray-500" with "border-white") but this was strangely inconsistent.

Upon referring to Tailwind's documentation, it turned out that this inconsistency was because its classes were only available to be hot swapped if they had already been used by an element once. The HTML file basically tells Tailwind which classes to include, and to save on resources, the rest are not included in the autogenerated CSS file. However, luckily Tailwind provide a configuration option for this issue called the "safelist". This array can be included in Tailwind's configuration file, and when filled with strings matching the name of Tailwind classes, tells Tailwind to include these in the autogenerated CSS, regardless of if they were initially found in the HTML or not.



Figure 78. MS24's light theme before the GUI rework

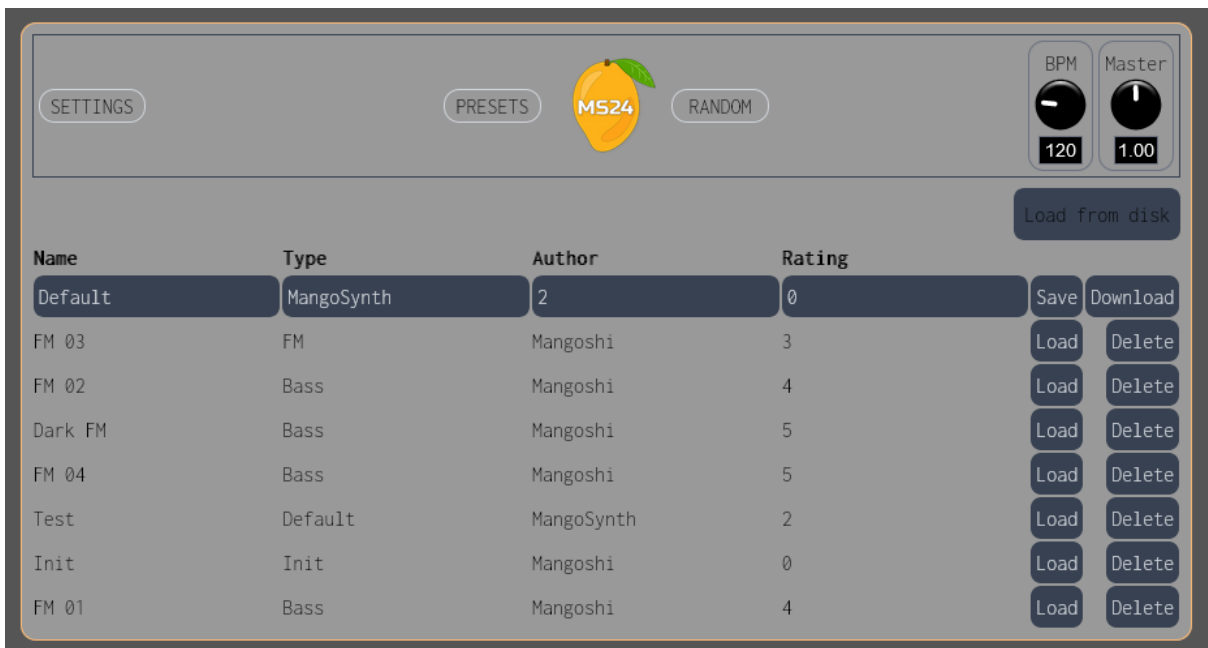


Figure 79. MS24's light theme before the GUI rework (presets page)

After filling the safelist array with the light theme alternatives to each class, a function was added to the JavaScript called "updateHtmlClasses" which takes two inputs, targetClass and newClass.



```
function updateHtmlClasses(targetClass, newClass) {  
  // Find all elements with targetClass  
  let targetElements = document.getElementsByClassName(targetClass)  
  // Loop through targetElements  
  while (targetElements.length > 0) {  
    // Add the new class  
    targetElements.item(0).classList.add(newClass)  
    // Remove the target class  
    targetElements[0].classList.remove(targetClass)  
  }  
}
```

Figure 80. The `updateHtmlClasses` function

This function first uses the `targetClass` input to create an array of all elements with that class. It then uses a while loop to go through each element individually, adding the `newClass` input to the list of classes, and removing the `targetClass`. Originally, this was attempted using a for loop, but this led to the strange issue of only some elements being updated when the theme was toggled, and each time it was toggled, it would result in different elements being updated than the last.

After a bit of research, it turned out that this was because each iteration of the for loop removing the `targetClass` would update the `targetElements` array's length, causing indexes to be skipped occasionally. For loops are usually used when the number of iterations is known, but in this case it wasn't. This is where a while loop comes in, which only stops once the condition is invalid.

Now that the function was working, it was time to put it to use. Twenty calls were added to the theme button's click event listener, ten for when switching to the light theme and ten for when switching to the dark one.

If switching to light theme, all dark borders and backgrounds were updated to lighter variants, while coloured borders, such as those used around buttons, were changed into background colours, so that the buttons would become filled in light mode, while only outlined in dark. When switching back, the reverse would occur, completing the updated theming functionality.



Figure 81. MS24's light theme after the GUI rework



Figure 82. MS24's light theme after the GUI rework (presets page)

5.9.2.3 Visualisations

The last part of the GUI rework would be visualisations. Up until this point, there had only been oscilloscopes added for the three oscillators in the synthesizer, with the filter, LFO, and FX units all containing empty visualisation areas. It didn't look great having all that empty space, so it was time to fix that.

Starting with the easiest of the lot, the effects unit would have been extremely difficult to design visualisations for, especially since there were multiple effects available, each with their own parameters. Rather than design nine individual visualisers, it made the most sense to just create another oscilloscope here, using the master output as its input. This would ensure that effects were

visualised despite which effect was selected and would double as a way to see how multiple waveforms interact with each other, even when the effects were off.

Next came the LFO. This one seemed like it would be quite tricky at first, as while an LFO is just an oscillator which oscillates at an extremely low frequency, visualisations for them often work a bit different to an oscilloscope, with the current value during oscillations usually being highlighted so that the modulation can be easier to see. However, it seemed like it might be possible to do with the existing oscilloscope code, at least after some changes.

Eventually, a rough implementation of the LFO visualiser was complete. The first changes required for it to work were to do with the Waveform object the LFO's output was fed into, parallel to the modulation target.

To start with, the Waveform itself had to be set at its maximum sample size of 16384 samples, compared to the 2048 samples the other Waveforms were using (which had originally been 1024, as that is the default). This was because having such a low oscillation frequency meant that a higher resolution was required to be able to draw enough lines between each sample for it to be equally as smooth as the others, once it came to visualising it.

The next difference with the Waveform was that Tone's Scale object was required, which performs linear scaling on an input signal. This Scale object was set to "-0.0001, 0.0001", which meant that if 0 was passed to it, it would return -0.0001, and if a 1 was passed to it, it would return 0.0001, with any numbers in-between being set within that range linearly. This was done because after testing the output of each Waveform, the LFO's Waveform had much higher values than the other oscillators, and they were all positive values, while the other oscillators naturally oscillated between negative and positive values each cycle. This was because LFO oscillations are determined by its min and max properties. Since min is set to 0 by default, and max is set to 1000, the LFO would oscillate between these two values at a rate determined by its grid setting, making it incompatible with the oscilloscope logic which uses 0 to determine the "zero-crossing", the point where a function changes from positive to negative (or visa-versa), to determine the starting point of the lines used to draw the signal. By using Tone's Scale object, it was possible to convert the LFO's output into one that was just like the other oscillators.

Now that the Waveform was outputting a similar range of values to the other oscillators, it was time to figure out how to visualise it using P5 and a slightly different approach, to better communicate the modulation while it was happening, since the logic used for the other oscillators resulted in a static waveform, only moving when the frequency or shape was updated, rather than throughout oscillations.

To do this, the original logic used to decide the start point of the visualised waveform was altered slightly so that if the target waveform was the LFO's, it would change the start point on every iteration, rather than only once the zero-crossing had been found. This was done by checking to see if the last waveform buffer value was equal to the current one. If not, the start point would be set the current one, resulting in a waveform that moves along the x-axis of the visualisation area while it oscillates, rather than staying in one place, anchored by the zero crossing.

With this logic, the LFO oscilloscope visualised the modulation effect quite well, as the rises and falls throughout an oscillation could now be followed, which matches the values being modulated. An unexpected side effect of this logic was that changing the min and max values would also update the visualisation, showing the depth of the modulation at any moment. This was both good and bad, as while it was nice to see under normal circumstances, if the LFO target is switched from the filter's

frequency to an oscillator's volume, the min and max control ranges are reduced significantly, so that they lie within the oscillator's volume range of -10 to 10, rather than 0 to 10,000. This means that depending on the LFO modulation target, the waveform drastically changes in height, with each rise and fall being almost invisible using such a narrow range.

```
for (let i = 1; i < buffer.length; i++) {
  if (wave !== lfo_waveform) {
    // if the previous point is negative, and the current point is positive/zero,
    // then we've found the zero crossing
    if (buffer[i - 1] < 0 && buffer[i] >= 0) {
      // Set the start point to the zero-crossing
      start = i
      break
    }
  } else {
    // since we want the oscilloscope to move with the waveform,
    // we assign the start to each unique buffer value
    if (buffer[i - 1] !== buffer[i]) {
      // Set the start point to the last unique value
      start = i
      break
    }
  }
}
```

Figure 83. Updated oscilloscope logic

With a bit of work, this unintended effect could be remedied by changing the gain of the waveform depending on the LFO target, but this wasn't attempted during Sprint 7, as the other items were prioritised.

The last of the missing visualisations was the filter. This was left until last because it was going to be the most difficult of the lot. Visualising how a filter works would consist of dynamically adjustable cubic Bezier curves, since there are three factors that each influence the shape of the filtering algorithms. The first of these would be the cutoff frequency. This is the point at which frequencies start to be filtered out. Changing this would move the visualised slope horizontally, across the x-axis. Next would be the resonance of the filter, which would be visualised as a peak or trough on the y-axis, at the cutoff point, as resonance emphasizes or attenuates the frequencies around this point. The last would be the rolloff, which is a slope that determines how quickly the frequencies past the cutoff point are attenuated.

This can be seen in Figures 84 and 85 below, which use screenshots of the popular VST, Serum, to demonstrate an implementation of a filter visualizer. In Figure 84, the cutoff, resonance, and rolloff factors have been annotated to help with understanding the problem at hand.



Figure 84. The Serum VST with the filter visualizer highlighted in red.

In the screenshot above (Figure 84), the filter visualiser has been highlighted. The filter parameters aren't visible since they only appear when changing values with the mouse, but we can see that the filter algorithm being used is an "MG Low 12", meaning it's a low-pass filter with a -12 decibel rolloff curve. This is quite a gradual slope, compared to the -24 and -48 dB rolloffs that can be chosen alternatively. The rolloff also influences the resonance to some degree. If a steeper slope was selected, the resonance peak (see Figure 84) would end up a little higher than with the -12 dB slope in use here.

It's worth mentioning that there is one extra parameter in Serum which is influencing the visualisation shown: the "drive" control. This control is used to amplify the unfiltered portion of audio, which is the reason why the line left of the resonance peak is a little higher than the zero-crossing, which is the default volume of the signal. If the drive control was at zero, the line left of the peak would return to zero from the resonance peak much more quickly.

It was eventually decided that at this late in the project, it would be too risky to attempt this visualiser. Doing so would require a decent understanding of Bezier curves, as well as the math behind each parameter of Tone.js' Filter objects, so that accurate curves could be visualised.

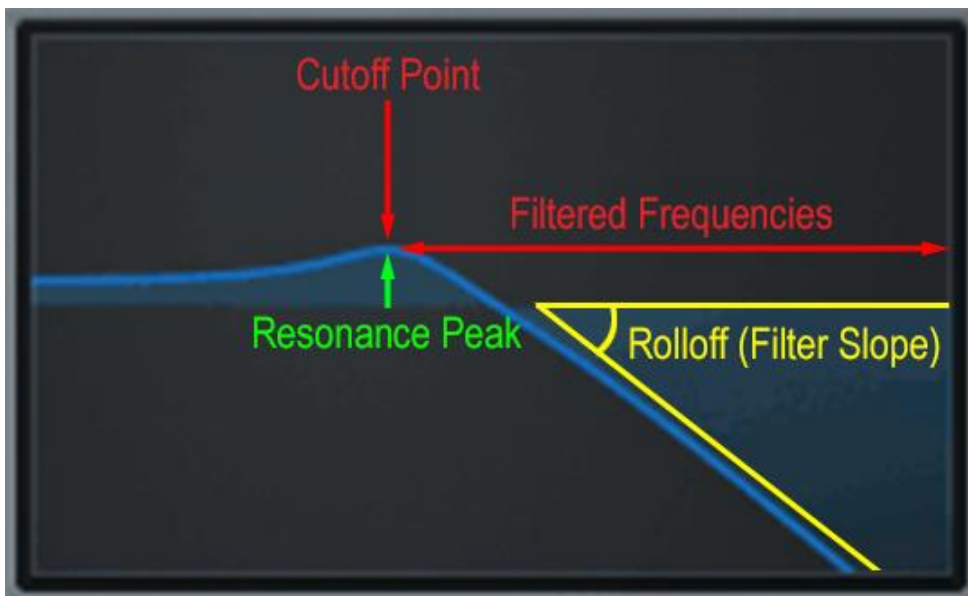


Figure 85. Serum's filter, annotated.

Instead of visualising the filter, later in the Sprint tooltips would be added to the otherwise empty visualization area, taking direct inspiration from another VST called Digitalis (Figure 86), in which an area of the effect's GUI is dedicated to a little cat called Jon, who gives hints depending on what the user's mouse cursor is hovering over. This would kill two birds with one stone, as tooltips would be difficult to implement in the remaining time if using the classic floating textbox approach.

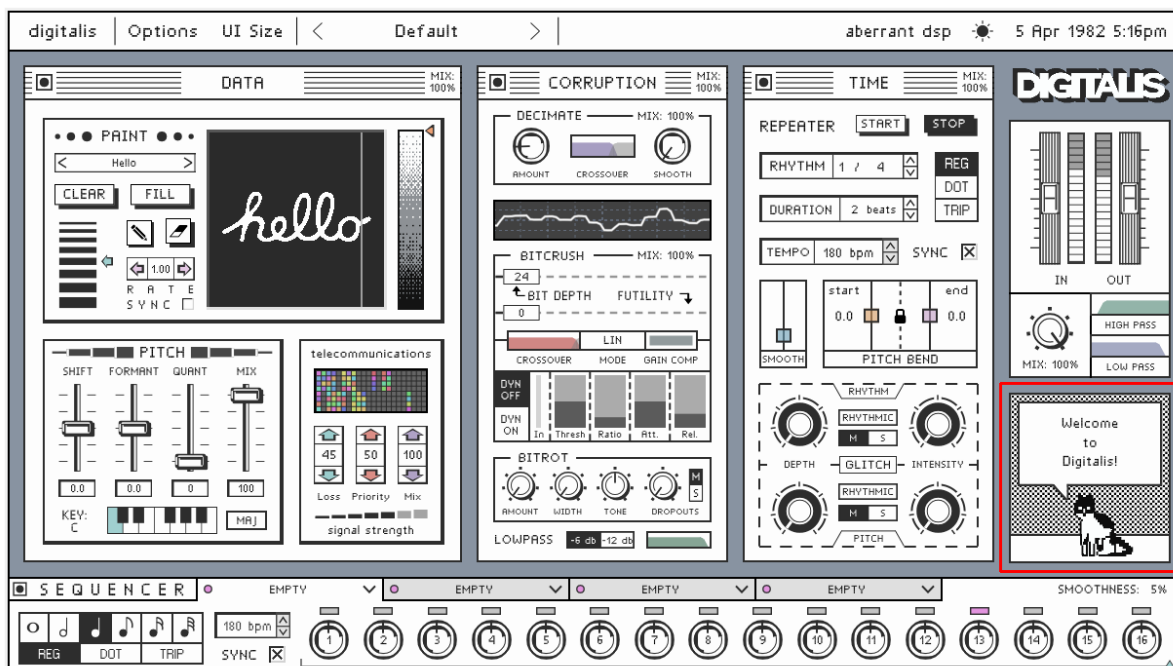


Figure 86. The Digitalis VST with its hint-giving companion Jon, highlighted in red.

5.9.3 Item 3: Randomization

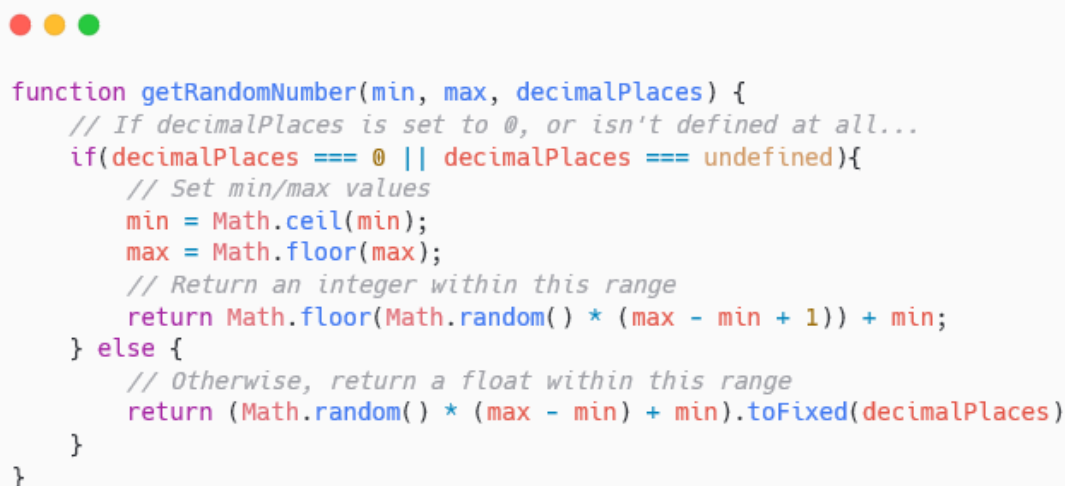
The third item worked on during the Sprint was preset randomization. The idea was simple, if a user clicks the “Random” button at the top of the interface, every control on the synth gets randomized, allowing for an easy source of inspiration or spontaneous sound design, should one desire it.

Preparations had been made for this feature in an earlier Sprint, when a JavaScript object called “MIN_MAX” was created, which holds an array of two values for every adjustable parameter on MS24, the first being the minimum value allowed, and the second being the maximum. All that was left to do was add functionality which used this object to randomize each parameter at once.

5.9.3.1 Functionality

The implementation of the functionality wasn’t too difficult. To start with, the object containing the minimum and maximum values needed to be adjusted to match any parameters whose ranges had been changed since it was originally created, and any new parameters such as arpeggiation controls had to be added to it as well.

Next came writing the logic responsible for the actual randomization. First up, a function needed to be created to generate a random number within a specified range, with a specified number of decimal places, as some of the synthesizer’s controls only accept integers, while others accept both integers and floats.



```
function getRandomNumber(min, max, decimalPlaces) {  
  // If decimalPlaces is set to 0, or isn't defined at all...  
  if(decimalPlaces === 0 || decimalPlaces === undefined){  
    // Set min/max values  
    min = Math.ceil(min);  
    max = Math.floor(max);  
    // Return an integer within this range  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
  } else {  
    // Otherwise, return a float within this range  
    return (Math.random() * (max - min) + min).toFixed(decimalPlaces)  
  }  
}
```

Figure 87. The function used to generate random integers or floats.

This function was then used to create a new preset when the “Random” button was clicked, by initialising a massive object with a key for every control on the synth, the value of each key being determined by the function, using the MIN_MAX object’s values as its inputs. See below for an example of this, in which an object representing the first oscillator is generated (Figure 88).

```

SYNTH_A: {
  enabled: getRandomNumber(MIN_MAX.OSCILLATOR.enabled[0], MIN_MAX.OSCILLATOR.enabled[1]),
  octave: getRandomNumber(MIN_MAX.OSCILLATOR.octave[0], MIN_MAX.OSCILLATOR.octave[1]),
  detune: getRandomNumber(MIN_MAX.OSCILLATOR.detune[0], MIN_MAX.OSCILLATOR.detune[1]),
  volume: getRandomNumber(MIN_MAX.OSCILLATOR.volume[0], MIN_MAX.OSCILLATOR.volume[1]),
  shape: getRandomNumber(MIN_MAX.OSCILLATOR.shape[0], MIN_MAX.OSCILLATOR.shape[1]),
  attack: getRandomNumber(MIN_MAX.OSCILLATOR.attack[0], MIN_MAX.OSCILLATOR.attack[1], 3),
  decay: getRandomNumber(MIN_MAX.OSCILLATOR.decay[0], MIN_MAX.OSCILLATOR.decay[1], 3),
  sustain: getRandomNumber(MIN_MAX.OSCILLATOR.sustain[0], MIN_MAX.OSCILLATOR.sustain[1], 3),
  release: getRandomNumber(MIN_MAX.OSCILLATOR.release[0], MIN_MAX.OSCILLATOR.release[1], 3),
  count: getRandomNumber(MIN_MAX.OSCILLATOR.count[0], MIN_MAX.OSCILLATOR.count[1]),
  spread: getRandomNumber(MIN_MAX.OSCILLATOR.spread[0], MIN_MAX.OSCILLATOR.spread[1]),
  harmonicity: getRandomNumber(MIN_MAX.OSCILLATOR.harmonicity[0], MIN_MAX.OSCILLATOR.harmonicity[1]),
  modulationIndex: getRandomNumber(MIN_MAX.OSCILLATOR.modulationIndex[0], MIN_MAX.OSCILLATOR.modulationIndex[1]),
  modulationShape: getRandomNumber(MIN_MAX.OSCILLATOR.modulationShape[0], MIN_MAX.OSCILLATOR.modulationShape[1])
}

```

Figure 88. An example of how “getRandomNumber” was used.

After generating the object, two switch cases were used to set the FX and LFO parameters, based on the randomly selected FX and randomly selected LFO modulation target. See below for an example of this (Figure 89).

```

switch(newPreset.FX.type) {
  case "Distortion":
    newPreset.FX.param1 = getRandomNumber(MIN_MAX.FX_DISTORTION.intensity[0], MIN_MAX.FX_DISTORTION.intensity[1])
    newPreset.FX.param2 = getRandomNumber(MIN_MAX.FX_DISTORTION.oversample[0], MIN_MAX.FX_DISTORTION.oversample[1])
    newPreset.FX.param3 = getRandomNumber(MIN_MAX.FX_DISTORTION.mix[0], MIN_MAX.FX_DISTORTION.mix[1])
    break
  case "Chebyshev":
    newPreset.FX.param1 = getRandomNumber(MIN_MAX.FX_CHEBYSHEV.order[0], MIN_MAX.FX_CHEBYSHEV.order[1])
    newPreset.FX.param2 = getRandomNumber(MIN_MAX.FX_CHEBYSHEV.mix[0], MIN_MAX.FX_CHEBYSHEV.mix[1])
    break
}

```

Figure 89. An example of how FX parameters were randomized based on the selected FX.

Once the switch case logic had been computed, all that was left to do was load the preset. Luckily, this functionality already existed, as it was used to load presets. By writing **loadPreset(newPreset)**, the newly generated synthesizer configuration would be loaded, setting every parameter across the synth, and updating the GUI to match.

5.9.3.2 The Problem

While this logic worked, there was an unforeseen outcome of preset randomization which wasn’t realised until experimenting with the newly implemented feature. This outcome was that preset randomization can be *extremely dangerous*.

Synthesizers can create a myriad of sounds, they can make beautifully delicate, soft sounds, or they can create *harsh* noise, sounding frequencies that are almost painful to one’s ears, and if not careful when tweaking parameters, at volumes that could really damage one’s ears.

Until this point, it wasn’t clear what parts of the synthesizer should be treated with caution. There are certain controls which should always be adjusted carefully, such as the resonance of a filter, but

dangerous noise is usually quite easy to avoid, as normally controls are incrementally adjusted, most people want to hear what a control does to a sound as its gradually changed, especially if new to synthesizers.

What the randomization button shone light on, was that there were more dangerous controls than previously thought. After one click of the random preset button, holding a key for a fraction of a second could sometimes result in a painfully loud sound, screeching at a high frequency, and there was no guarantee it would stop once the key was released, because the ADSR envelope might have been set with a long release, or maybe a delay effect had been enabled with its feedback control pushed to the maximum, resulting in a feedback loop and volumes that would ramp up until the synth's internal limiter was activated, which could easily be after damage had been done to the user's speakers, headphones, or worse-case, their ears.

There were two ways to go about this issue. One was to manually go through each control, shifting the minimum and maximum values into safe ranges so that the randomization didn't risk creating dangerous configurations. The second was to completely remove the randomization button and functionality.

Since a feature branch was being used for this, it was left unmerged for the time being. It was simply too dangerous to include in its current state. No matter how many warning messages could be added to alert a user to the danger, it just didn't feel right. While avid synthesizer users would be aware of these dangers, a beginner would not, and since MS24 would be a lot more accessible than most synths out there, there would likely be more beginners using this application than using a VST imported into a DAW.

5.9.4 Item 4: Tooltips

The last item in Sprint 7 was the implementation of tooltip functionality. This was deemed important as the majority of users tested in Sprint 6 were unsure of what each control on the interface was for, requiring trial and error to understand in most cases. As mentioned in Item 2, it was decided that instead of a filter visualisation, tooltips could be contained within the visualisation area, acting like the area Jon the cat from the VST Digitalis resides in (Figure 86).

When using Digitalis, if you want to know what a certain element of the interface does, all you need to do is hover of that element with the mouse cursor, and Jon the cat's speech bubble gets filled with text, describing what that element does and how to use it. This is quite a nice approach, as the tips are always there while using controls. There's no need to place the cursor over an element and wait for the tooltip to pop up, like on Windows, nor is there any need to open a guide externally; if you want to know what something does, Jon is always there to tell you.

5.9.4.1 Functionality

To implement tooltips functionality, mouse-based event listeners would need to be tied to every element a tooltip was deemed necessary for, before text within the visualisation area could be dynamically injected, based on which element was being hovered over.

To start with, the tedious process of adding a class to each element that felt like it needed an explanation began. This class was "tooltipElement" and was added to approximately 87 elements. The canvas element that had been placed in the filter's visualisation area was then replaced with a

div containing two paragraph elements, the first containing bold text explaining “Tooltips go here!” and the second containing regular text, instructing to “Hover over something to see what it does”.

After this, the JavaScript was updated. Starting with the tooltips themselves, a new object was created called “tooltips”, which was filled with a list of objects representing each type of control. Each of these objects were filled with yet more objects, representing the individual controls. Each of these were given two keys, “top” and “bottom”, whose values would be Strings containing the tooltips themselves.

```
let tooltips = {
  Defaults: {
    top: "Tooltips go here!",
    bottom: "Hover over something to see what it does!"
  },
  // Buttons (Settings, Presets, Random)
  Buttons: {
    Settings: {
      top: "Change synth settings",
      bottom: "Return home, or change the theme!"
    },
    Presets: {
      top: "Open the preset menu",
      bottom: "Load a preset, or save your own!"
    },
    Random: {
      top: "Randomise the synth settings",
      bottom: "Careful! This will overwrite your current settings!"
    }
  },
  // And so on...
}
```

Figure 90. The first version of the tooltips object.

Once the tooltips object had been created, it was time to tie event listeners to each element with the tooltipElement class, so that it was possible to tell which element was being hovered over with the mouse cursor. The tooltip top and bottom paragraph elements were initialised in the JavaScript using the getElementById() method, while getElementsByClassName() was used to create an array containing every element with the class tooltipElement.

A for loop was then created to loop through each of the elements in the array. It was realised by this point that the best way to go about this would be to attempt to use the ID and tag name of each element to dynamically retrieve the correct tooltip from the tooltips object. Until now, the object was using keys that described the elements but didn't match their names exactly. For example, as can be seen in Figure 90, the tooltips for the settings button were located in a “Settings” object, nested inside a “Buttons” object.

If this was adjusted so that “Buttons” became “button”, matching the tag name <button>, and if “Settings” became “settings_button”, matching the ID of the button, the for loop could be written in such a way that each element iterated over would use its own metadata to access the correct key/value pair in the tooltips object.

```

let tooltips = {
  defaults: {
    top: "Tooltips go here!",
    bottom: "Hover over something to see what it does &#128516;"
  },
  // Buttons (Settings, Presets, Random)
  button: {
    settings_button: {
      top: "Open settings menu",
      bottom: "Return home, or change the theme!"
    },
    presets_button: {
      top: "Open preset menu",
      bottom: "Load a preset, or save your own!"
    },
    random_preset_button: {
      top: "Randomise synth settings",
      bottom: "Careful! This can result in VERY loud sounds!"
    }
  },
  // And so on...
}

```

Figure 91. The updated version of the tooltips object.

After updating the tooltips object so that the nested objects at the top-level matched the tag names, and the ones at the bottom-level objects matched the IDs, the for-loop logic was returned to. At the beginning of the loop, the current element's ID and tag name are assigned to variables "element_id" and "element_tag". To get the element's tag as it looks in the HTML, the property localName needed to be accessed instead of tagName, as for some reason tagName returned the name in a different case to how it appeared in the HTML, at least in the case of the webaudio-controls knob element.

A "mouseover" event listener is then added to the element, which fires if the mouse cursor enters that element. Within this event listener, a variable called "tooltip" is initialised, before an if statement which checks to see if the first index of the element's ID, split by the underscore character, is equal to "osc", and if the element tag is equal to "webaudio-knob" or "webaudio-slider" at the same time. If these conditions are met, tooltip is assigned to the following:

```
tooltips[element_tag][element_id.split("_").splice(2).join("_")]
```

For this example, let's say we're hovering over oscillator A's volume knob. The line above accesses the tooltips object and uses the element's tag as the first key. The element's tag in this case is "webaudio-knob", so we are now accessing tooltips["webaudio-knob"].

It then uses the split method to split the element's ID into an array using the underscore character as a delimiter. Since the ID of this element is "osc_a_volume", we now have an array with the first index containing "osc", the second index containing "a" and the third index containing "volume".

Next, we use the splice method to cut out everything before the third index (which is 2, as arrays start at 0). Finally, we use the join method to join each element in the array back into a String using the underscore character as a separator between each element.

This returns "volume", meaning we're now accessing tooltips["webaudio-knob"]["volume"].

The reason for using the join method is so that the likes of “osc_a_fm_depth” becomes “fm_depth” rather than just “fm”. The reason for using this at all, is to cut down on repeated tooltips, as there are three oscillators with almost entirely identical controls. Rather than create identical tooltips for “osc_a_volume”, “osc_b_volume” and “osc_c_volume”, it made more sense to just create tooltips for “volume” within the “webaudio-knob” nested object.

```
let tooltip_top = document.getElementById('tooltip_top')
let tooltip_bottom = document.getElementById('tooltip_bottom')
let tooltip_elements = document.getElementsByClassName('tooltipElement')

// Loop through each element with the class "tooltipElement"
for (let i = 0; i < tooltip_elements.length; i++) {
  // Get the element's ID (i.e. "osc_a_volume")
  let element_id = tooltip_elements[i].id
  // Get the element's tag name (i.e. "webaudio-knob")
  let element_tag = tooltip_elements[i].localName
  // Add mouseover event listener to each button
  tooltip_elements[i].addEventListener('mouseover', function () {
    // Log the element's tag and ID
    console.log('mouseover', element_tag, element_id)
    // Get the tooltip text from the tooltips object
    let tooltip
    // If the first part of the ID is "osc", and the element is a knob or slider
    if (
      (element_id.split('_')[0] === 'osc' && element_tag === 'webaudio-knob') ||
      (element_id.split('_')[0] === 'osc' && element_tag === 'webaudio-slider')
    ) {
      // Log the ID without the first two parts (e.g. "osc_a_fm_depth" becomes "fm_depth")
      console.log(element_id.split('_').splice(2).join('_'))
      // Use the ID without the first two parts to get the correct tooltip text
      // This is to reduce the amount of duplicate tooltip values!
      tooltip = tooltips[element_tag][element_id.split('_').splice(2).join('_')]
    } else {
      // Otherwise, just use the ID to get the correct tooltip text
      tooltip = tooltips[element_tag][element_id]
    }
    // Set the tooltip text
    tooltip_top.innerHTML = tooltip.top
    tooltip_bottom.innerHTML = tooltip.bottom
  })
  // Add mouseout event listener to each button
  tooltip_elements[i].addEventListener('mouseout', function () {
    console.log('mouseout', element_tag, element_id)
    // Return the tooltip text to the default
    tooltip_top.innerHTML = tooltips.defaults.top
    tooltip_bottom.innerHTML = tooltips.defaults.bottom
  })
}
```

Figure 92. The tooltip event listening logic.

The else of this if statement needs much less explanation, as it simply uses the element ID as the second key, with no modification.

After the correct tooltip has been found, the innerHTML property of the tooltip_top and tooltip_bottom paragraph elements is used to update the text inside.

A “mouseout” event listener is also added to the element being iterated over, which gets triggered as soon as the mouse cursor leaves the element. Once this happens, the `tooltip_top` and `tooltip_bottom` paragraph elements are set back to their default values, informing the user that they can hover over elements to see hints.

While this was enough for most elements to work, there were still 26 tooltips of the 113 in total that weren’t accessible yet. This was because they were explanations for each of the different effects parameters, which aren’t part of the HTML until the selected FX is changed.

The tooltips for these parameters were moved to a different object called “dynamicTooltips”. The logic responsible for switching effects in and out was then updated so that depending on the selected effect, the tooltips object’s FX tips would be updated to match the tip from `dynamicTooltips`.

For example, if the Reverb effect was selected, `tooltips[“webaudio-knob”].fx_param1` would be set to `dynamicTooltips[“Reverb”][“fx_param1”]`, `tooltips[“webaudio-knob”].fx_param2` would be set to `dynamicTooltips[“Reverb”][“fx_param2”]`, and so on. This meant that regardless of the effect selected, each of the knobs would show a tooltip specific to that effect’s parameters.

With this functionality complete, the application now felt a lot more user-friendly, especially toward beginner synthesizer users, who might not understand most of the controls available. Rather than learning by trial and error, the tooltips would allow users gain a better understanding of what they’re adjusting, while also warning of potentially dangerous controls, such as the filter resonance knob, which was given a “be careful!” at the end of the tooltip, as high resonance values can sometimes result in screechy sounds.

5.10 Sprint 8

The goal for this Sprint was to finalize the application and this thesis, by making any finishing touches deemed necessary to bring them to completion.

5.10.1 Finishing Touches

The finishing touches made to the code included updating the welcome screen with an image of a keyboard with all the computer keyboard keys that can be used to play notes annotated for each key. Adobe Photoshop was used for this. The welcome screen text was also improved, by making the guide more concise and by removing redundant bug warnings.

Next, the default presets that MS24 loads upon launch were removed and replaced with ten new ones, since the old ones were mostly broken after the last few major updates to the synth logic. This consisted of running the application and configuring the presets manually, while trying to create a wide range of sounds, to demonstrate what MS24 is capable of.

Following this, minor fixes were implemented, such as storing the currently selected theme in `localStorage` so that reloads would not reset the theme, removal of the randomization button since the logic was disabled, and removal of the broken LFO modulation targets.

Next, redundant code was removed, refactoring was done where necessary, and comments were added where missing. The application was then built, and the output dist folder was copied into the MS24 Electron directory. The Electron project was then launched in WebStorm and used to build the Windows 64-bit binary. This binary was then included in the main repository so that it could be included in the official release.

Finally, the README was updated to include instructions for use, and this thesis was added to the repository. GitHub's release functionality was then used to package the application into an official release, containing the archived source code and binary file.

5.11 Conclusion

In summary, this chapter focused on the implementation of MS24. The chapter begins with an overview of the project, the technologies and development environment used to build the application, as well as the Scrum methodology followed, splitting the fifteen weeks of development into seven two-week long Sprints, followed by a final one-week Sprint.

The chapter then delves into the implementation process for each Sprint, describing in detail the goal of the Sprint versus the work completed, split into Items. Along the way, code snippets are used to help explain the more complex parts of the application's code, as well as screenshots of the application and several applications that inspiration was taken from when designing some of the features.

6 Testing

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

1. Functional Testing
2. User Testing

Functional testing is a type of software testing whereby the system is tested against the functional requirements set out during the requirements gathering stage. The app is tested by looking to see if the actual output for a given input corresponds with the expected output.

User testing is used to ensure that the software meets the needs and expectations of the end-users. It provides valuable feedback on the user experience, identifies usability issues, and helps to improve the overall quality of the software product. By incorporating user feedback during development, the final product is more likely to be successful and meet the needs of the target audience.

6.1 Functional Testing

This section describes the functional tests which were carried out on the app. Each table focuses on a specific category of functionality, with each row including a description of the test case, the input, expected output, actual output, and result of the test.

6.1.1 User Input

Test No	Description of test case	Input	Expected Output	Actual Output	Result
1	PC keyboard input	Keys pressed	Notes play	Notes play	PASSED
2	Mouse input (keyboard)	Keys clicked	Notes play	Notes play	PASSED
3	Mouse input (drag)	Control dragged	Control changes	Control changes	PASSED
4	Mouse input (scroll)	Control scrolled	Control changes	Control changes	PASSED
5	MIDI keyboard input	Keys pressed	Notes play	Notes play	PASSED
6	MIDI CC input	MIDI learn, CC input	Control changes	Control changes	PASSED

6.1.2 Synthesizer Functionality

Test No	Description of test case	Input	Expected Output	Actual Output	Result
1	Generates tones using oscillators	Key input	Tones generated	Tones generated	PASSED
2	Oscillator wave-shaping	Controls adjusted	Waveform modified	Waveform modified	PASSED
3	Frequency Modulation (FM)	Controls adjusted	Frequency modulated	Frequency modulated	PASSED
4	ADSR Envelopes	Key input	Amplitude enveloped	Amplitude enveloped	PASSED
5	Filter routing	Filter enabled, key input	Generated tones get filtered	Generated tones get filtered	PASSED
6	Effects routing	Effects enabled, key input	Effects applied to generated tones	Effects applied to generated tones	PASSED
7	LFO modulation	LFO enabled, key input	Parameter modulated by LFO oscillation	Parameter modulated by LFO oscillation	PASSED
8	Arpeggiation	Arpeggiator enabled, key input	Arpeggio generated from held keys	Arpeggio generated from held keys	PASSED
9	Default presets	Synth loaded	Presets list populated	Presets list populated	PASSED
10	Preset saving	Preset save button click	Preset saved to localStorage	Preset saved to localStorage	PASSED
11	Preset saving (to database)	Preset save button clicked	Preset saved to database	Preset saved to localStorage	FAILED
12	Preset downloading	Preset download button click	Preset downloaded to computer	Preset downloaded to computer	PASSED
13	Recording & exporting	Recorder toggled	Sound recorded and exported	Sound recorded and exported.	PASSED
14	GUI themes	Theme button clicked	Theme changed	Theme changed	PASSED
15	Oscilloscopes	Key input	Waveforms visualised	Waveforms visualised	PASSED

6.1.3 Discussion of Functional Testing Results

All but one of the functional tests passed. The test which failed was the ability to save presets to a user account, hosted on a database so that any user's presets could be accessed from anywhere. The reason the test failed is because a database and server were never developed for the application, meaning that presets can only be saved locally, to the user's browser or downloaded to their computer.

6.2 User Testing

To properly test the usability of MS24, five unmoderated user tests were conducted in-person using friends and family as participants, since the educational license for the online tool Loop11 originally planned to be used for remote testing of as many users as possible had expired, and there was no time to wait for this to be resolved.

Loop11 did allow the test to be created using the hosted version of MS24 and allowed tasks and questions to be added and previewed without issue. It wasn't until the test design had been completed and the "launch test" button was clicked however, that it notified of the expired license.

Instead of scrapping the tasks and questions created, they were taken down and were decided to be used for in-person usability testing instead.

Each test consisted of five tasks and eight questions, shown below:

Task #1: Read the instructions before granting access to the Web Audio API.

Task #2: Use your computer keyboard to play some notes on the synthesizer.

Task #3: Enable the sub oscillator and apply the distortion effect.

Task #4: Start recording, play some notes, then stop recording.

Task #5: Open the presets menu and load a preset.

Likert Question #1: The welcome screen instructions were clear and helped me complete the tasks.

Likert Question #2: Using the computer keyboard to play notes was easy and intuitive.

Likert Question #3: Enabling the sub-oscillator and changing its rate was easy and intuitive.

Likert Question #4: Toggling the audio recorder on and off was straightforward.

Likert Question #5: The presets menu was well laid-out and intuitive.

Likert Question #6: Loading a preset was straightforward.

Likert Question #7: I liked the design of MS24's graphical user interface (GUI).

Open-ended Question #1: Do you have any feedback, comments, or suggestions?

For each test, the participant was given a laptop with the hosted version of the application open on the browser, alongside the list of tasks. Once complete, each user was asked to enter their answers on an Excel spreadsheet. After each of the tests had been completed, the data was compiled, and pie charts were generated for each set of Likert Scale answers. These pie charts can be found below.

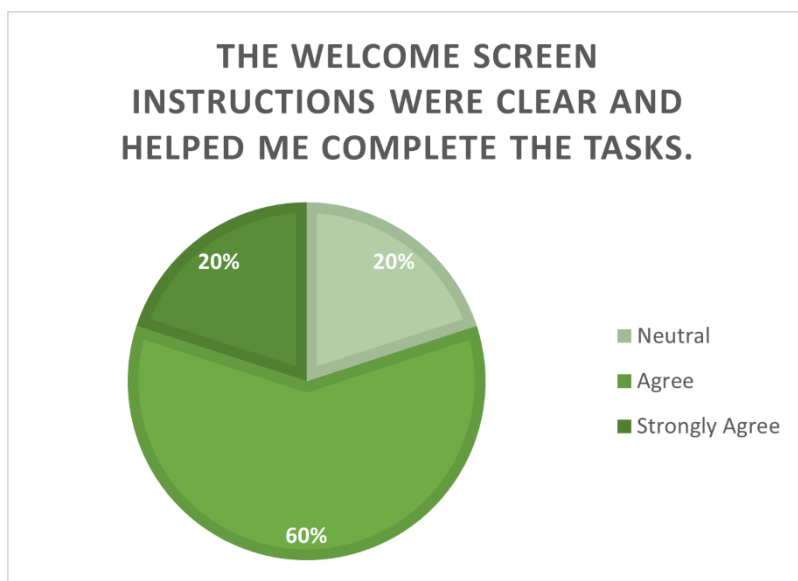


Figure 93. Likert Scale answers for question 1.

Most participants agreed that the welcome screen instructions were clear, helping them to complete the rest of the tasks. However only one strongly agreed with this, and one wasn't sure. The welcome screen might need a few improvements to provide better information and instructions.

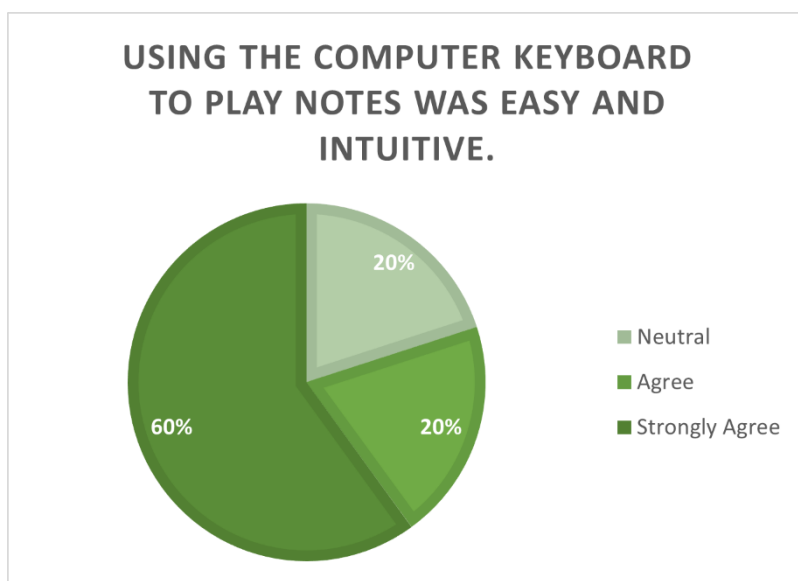


Figure 94. Likert Scale answers for question 2.

Most participants strongly agreed that using the computer keyboard to play notes was easy and intuitive. The users who were unsure probably had a hard time remembering the keyboard instructions from the welcome screen, since they are text-based and not visual.

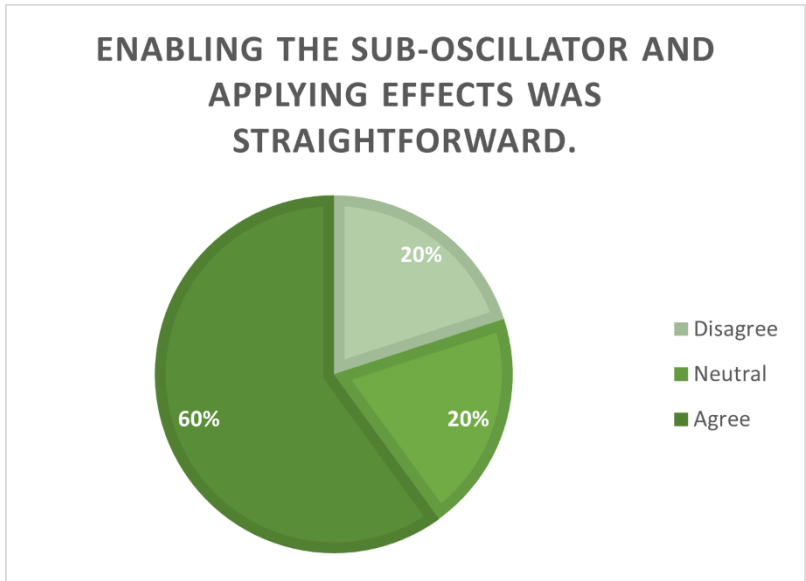


Figure 95. Likert Scale answers for question 3.

Most participants agreed that enabling the sub-oscillator and applying distortion effects was straightforward. However, with one neutral and one disagree, better instructions may be required to ensure all users have a better understanding of what control does what.

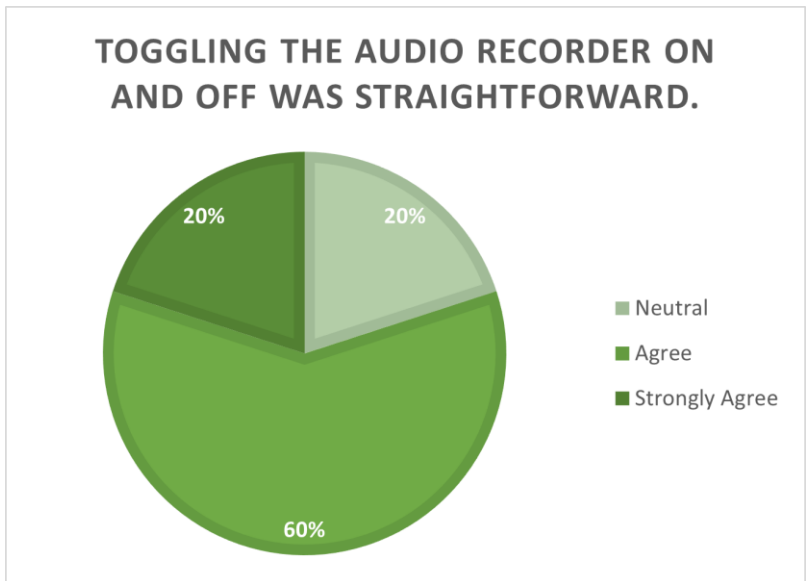


Figure 96. Likert Scale answers for question 4.

Most participants agreed that audio recording was straightforward. However, like with the previous tasks, this feature could be better explained.

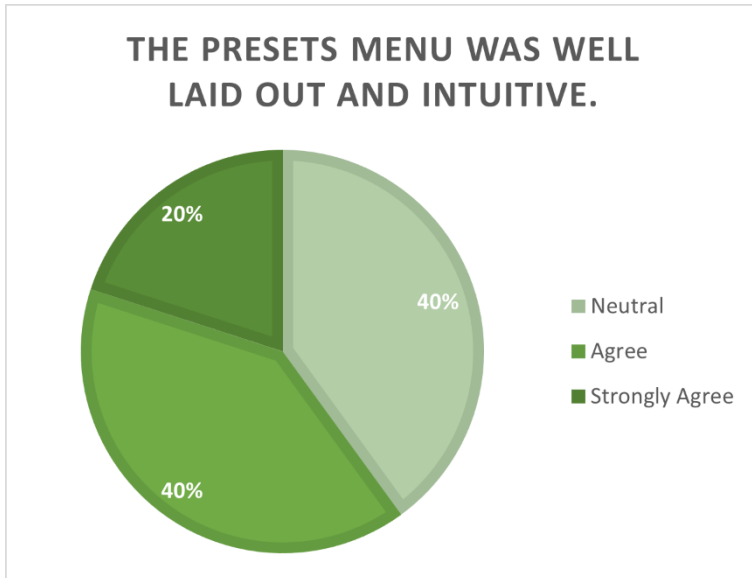


Figure 97. Likert Scale answers for question 5.

Participants were a little less comfortable with the presets menu. This was understandable since it wasn't finished at the time of testing and didn't look great. This would have to be corrected for the final version of the app.

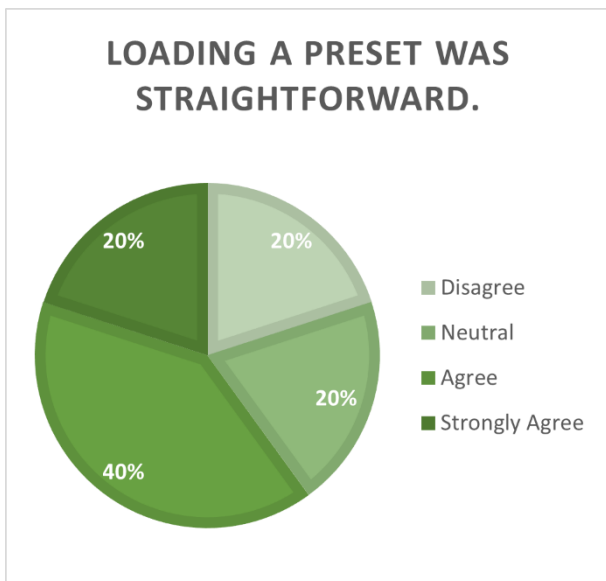


Figure 98. Likert Scale answers for question 6.

Participants were a bit happier when it came to loading presets, but this was probably because once they found the presets menu it was easy enough to find the load buttons. This question didn't make much sense in hindsight.

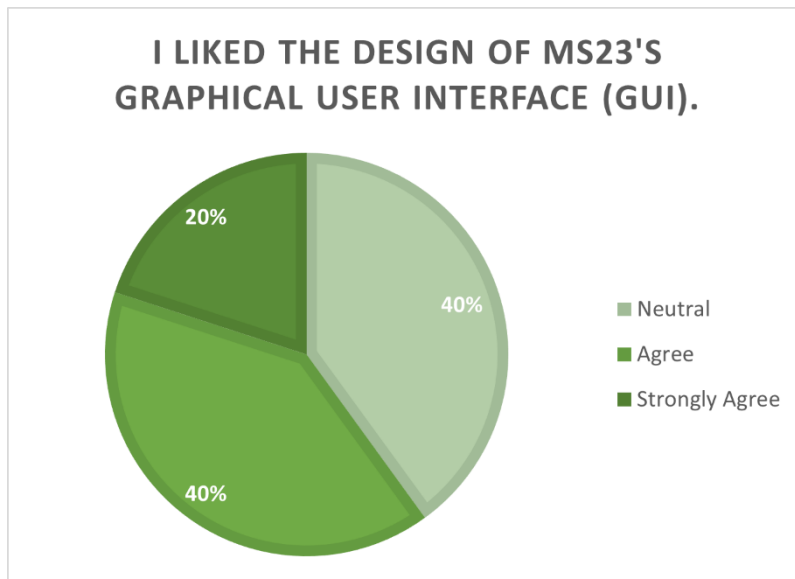


Figure 99. Likert Scale answers for question 7.

Most participants liked MS24's GUI overall, but two participants were unsure. This was likely because at the time of testing, some of the interface was hidden as the GUI was too big for the browser's viewport but might also have been due to other factors.

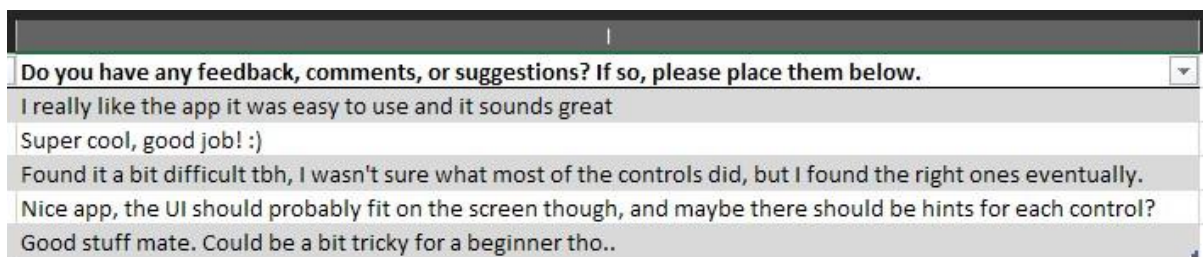


Figure 100. Open-ended answers for the final question.

The final question was left open-ended, to allow users to give their thoughts in their own words. Two of these answers were purely positive and gave no criticism, while the other three each provided a bit of criticism.

The first of these stated that they found it a bit difficult, as they weren't sure what most of the controls did, stating they did eventually find the right ones, but it must have taken a lot of trial and error to do so.

The second stated that they liked the app, but criticized the fact that some of the UI didn't fit on the screen and suggested that maybe hints could be given for each control, as they must have been a bit unsure of the controls too.

The last wasn't very specific, just saying that it "could be a bit tricky for a beginner".

With these results, it was clear that while the synthesizer was mostly fine, the user interface could do with a few improvements, specifically some way to communicate what each control does, so that beginner synthesizer users aren't forced to learn through trial and error.

These usability tests were the main reason for the GUI rework and implementation of tooltips functionality in Sprint 7, which sought to address the most common criticisms given by the usability test participants.

6.3 Conclusion

In conclusion, this chapter discussed the process of the testing phase of the application, starting with functional testing to compare the expected outputs of various inputs with the actual outputs, to see how well the application met the functional requirements. Following this, usability tests were conducted with five participants to see how well the application met the non-functional requirements.

Overall, the testing results were quite positive. With only the online capabilities of MS24 lacking, and some minor usability issues, the results of the testing phase would be referred to when finalising the application, so that as many issues found could be solved before the final release.

7 Project Management

This chapter describes how the MS24 project was managed by discussing the tools used to organise the tasks and codebase along the way, such as Notion for the Sprint Backlog, GitHub for the hosting and version control, GitKraken to help with GitHub functionalities, and WebStorm's TODO features for managing code-specific tasks within the IDE. Following this, a brief explanation of how the SCRUM methodology was incorporated into the management of the project is described.

7.1 Project Management Tools

7.1.1 Notion

Notion is a powerful, versatile project management and note-taking tool provided as a web application and desktop application. Notion was chosen to manage the Sprint Backlog for MS24 instead of its widely used alternative Trello, which provides a far more limited set of functionalities, restricting users to a Kanban-style board view.

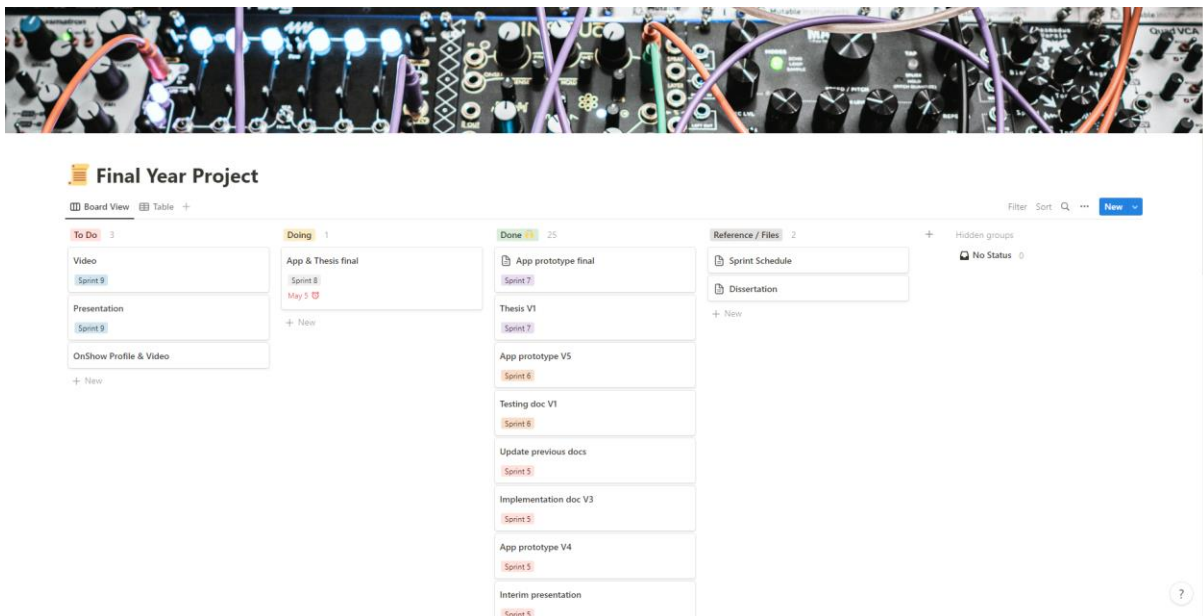



Figure 101. Notion's Kanban-style board view (MS24 project).

Each item on the Kanban board (Figure 101) can be opened so that extra detail can be added, if necessary, with functionalities like inline tables, embedded document, and markdown formatting available to style the data in any way that is desired. The Kanban view can also be swapped out with several other views, such as a table view (Figure 102), a timeline view, or a calendar view.



Final Year Project

Area	Date Created	Due	Sprint	Status
Video	March 6, 2023 6:26 PM		Sprint 9	To Do
Presentation	March 6, 2023 6:26 PM		Sprint 9	To Do
App & Thesis final	March 6, 2023 6:25 PM	May 5, 2023	Sprint 8	Doing
App prototype final	March 6, 2023 6:25 PM		Sprint 7	Done
Thesis V1	March 6, 2023 6:25 PM		Sprint 7	Done
App prototype V5	March 6, 2023 6:23 PM		Sprint 6	Done
Testing doc V1	March 6, 2023 6:24 PM		Sprint 6	Done
Update previous docs	March 6, 2023 6:22 PM		Sprint 5	Done
Implementation doc V3	March 6, 2023 6:22 PM		Sprint 5	Done
App prototype V4	March 6, 2023 6:22 PM		Sprint 5	Done
Interim presentation	March 6, 2023 6:23 PM		Sprint 5	Done
Design doc final	March 6, 2023 6:21 PM		Sprint 4	Done
App prototype V3	March 6, 2023 6:21 PM		Sprint 4	Done
Implementation doc V2	February 16, 2023 1:07 PM		Sprint 4	Done
Implementation Doc V1	February 13, 2023 5:57 PM		Sprint 3	Done
Design Doc V2	February 13, 2023 5:57 PM		Sprint 3	Done

Figure 102. Notion's table view (MS24 project).

In practice, Notion helped tremendously when it came to staying organised and keeping on top of the many items in the Sprint Backlog. As well as this, its notification system was used to set up alerts, ensuring that upcoming deadlines never came as a surprise.

7.1.2 Git & GitHub

GitHub is a widely used provider of hosting for open-source software development, utilizing the Git distributed version control system created by Linus Torvalds. Regardless of whether it's a solo developer or a large IT company, the safety net offered by distributed version control is hard to argue with. This system provides what could be seen as an advanced undo mechanism, enabling developers to track changes made to their code by committing their work as they progress. This allows them to refer to previous versions of their code in the event of errors, or even revert the codebase to previous versions by using the "checkout" functionality if need be.

GitHub was extremely helpful in developing MS24. It allowed every major change to the codebase to be committed and documented in detail, providing a long list of checkpoints that could be returned to if old logic needed to be referred to when implementing new logic, or when writing the implementation chapters of this thesis. By making the commit messages (used to document the changes) as detailed as possible, the implementation chapter was far easier to write, only requiring a read of the commit messages to know exactly what changed at what point.

Another way in which GitHub helped was through its branch functionality. Implementing the rework of the keyboard logic, as well as the GUI, consisted of many small changes which would leave the application broken until the implementation had been completed. As MS24 was hosted from early on, using the main branch of the tree for these changes would have temporarily made it unusable. Instead, feature branches were created for each major rework, separating the commits from the main branch, leaving the hosted version usable while also grouping the many individual commits together. At the end of each rework, a pull request was made, which provides a way to safely merge

the new codebase with the one on the main branch, while also documenting the changes in more detail with Markdown formatting.

7.1.3 GitKraken

A third-party desktop application called GitKraken was used to help with managing the repository instead of the first-party tool GitHub Desktop due to personal preference. The tool was mainly used to visualize the repository tree throughout the project and to compose detailed multi-line commit messages.

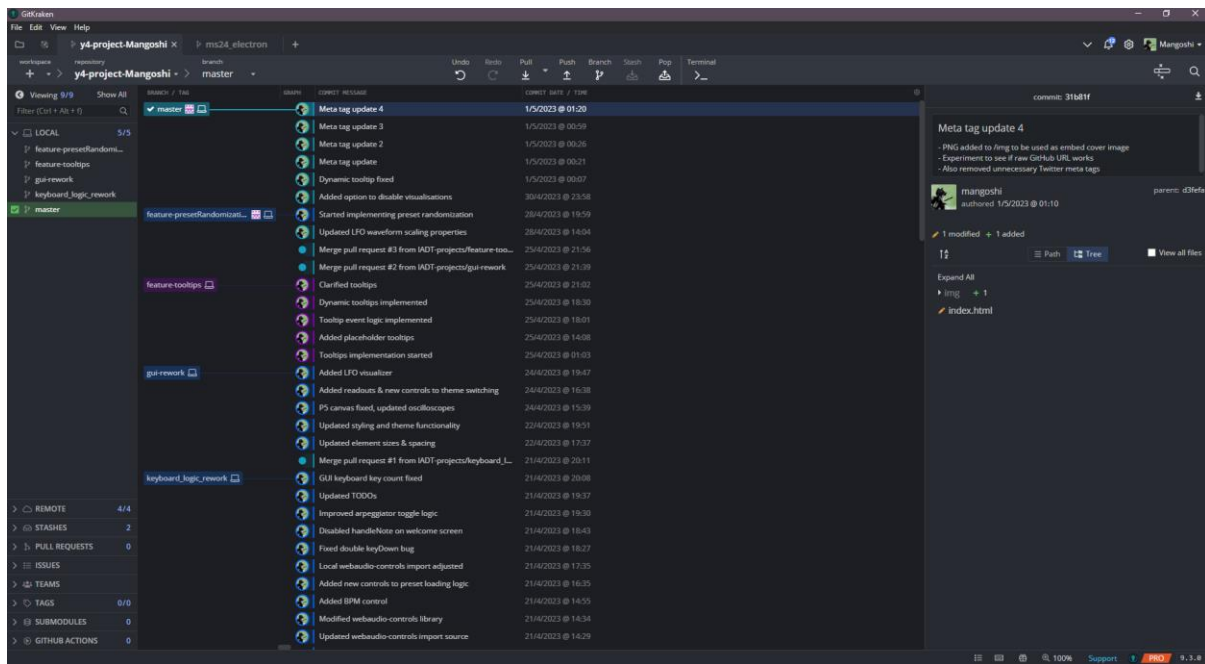


Figure 103. The GitKraken GUI

7.1.4 WebStorm TODO Comments

WebStorm, the IDE used to develop MS24, was also used to manage the project's tasks. This was done through its TODO comment functionality, in which comments starting with "TODO" become highlighted in a different colour to other comments and get added to a part of the IDE's window, allowing them to be clicked, jumping to that specific point in the code.

This was incredibly useful, as it allowed detailed to-do lists to be added directly to the code, with a list of general to-dos positioned at the top of the code, and one-liner to-dos to be added above any issues encountered, reducing the need to refer to other software such as Notion when figuring out what to work on, as everything was contained in the same place.



Figure 104. WebStorm's TODO comment section.

7.2 SCRUM Methodology

The SCRUM methodology is a popular framework for agile project management, commonly used in software development. It involves breaking down a project into small, manageable tasks which are completed in short “sprints” which usually last a few weeks.

While the SCRUM methodology is intended for large development teams, many of its principles can be incorporated by small teams or even solo developers, as was the case with MS24. By breaking down the project into small tasks known as items, the principles of SCRUM were able to help prevent becoming overwhelmed by what would otherwise be a huge list of tasks.

Each Sprint over the course of the implementation phase took place over two weeks, with goals for each Sprint being set before they began, by referring to the Sprint Backlog to see what blockers (items that hadn't been completed prior) and items were to be tackled. This allowed for a constant level of focus on the most important steps of development to be maintained, improving productivity and morale throughout the course of the project.

8 Project Reflection

8.1 Your views on the project

I thoroughly enjoyed working on MS24. It's hard to put into words how much of a passion music is to me, from listening to producing and everything in-between. As someone who's been practicing music production for the last ten years, and studying computing for the past five, it only made sense for my final project to be a combination of these things I love.

The idea didn't come easily though, throughout the summer leading up to fourth year I toyed with several ideas, from audio/visual experiences made with TouchDesigner, to more traditional audio software with Max MSP, it took a while for the idea of combining web development and music software to come to mind, but once it did, I knew it was the one.

Initially, the project was a bit stressful. Tone.js was a confusing beast to tame, with somewhat misleading documentation and many subtle but important intricacies, it took a long time to get my head around certain aspects of it, which greatly hindered development time at first. Combined with attempting to use React and the state management library Zustand, many issues I encountered were confusing, as I didn't know where the problem stemmed from. Switching to Vanilla JavaScript midway was a little disheartening at first, but it ended up simplifying the troubleshooting processes enough for me to learn Tone.js properly and figure out how to connect each of its components together to make the synthesizer I had planned.

Sadly, some of the puzzles presented by Tone have been left unsolved. Certain components such as the low-frequency oscillator responsible for modulating components really doesn't like to be connected and disconnected from other components as freely as you would expect based off the other components' behaviour, meaning MS24 has a lot left to be desired when it comes to modulation, an important tool in any sound designers' belt. But regardless of this, I'm delighted with how much I was able to figure out and am still so surprised by how many interesting sounds I can make with my application, as I really wasn't sure what to expect when starting the project.

Lastly, the feedback I've gotten from MS24, especially around the time I sent the survey out to multiple music production and record label Discord servers, has been so encouraging. It was really nice to see how many people out there seemed genuinely interested in the project. Even the developer SixthSample, who makes multiple VST effects I commonly use in my music showed great interest, and offered his hand, telling me to let him know if I got stuck along the way, despite the technologies being worlds apart. The whole experience has made me interested in trying this kind of thing again, or at very least in continuing to develop MS24 in my own time over the next few years, to see how much better it can get.

8.2 Completing a large software development project

The process involved in completing a large-scale software development project is certainly intense, requiring efficient time management and project management to stay on top of things. However, in a lot of ways its far more rewarding to finish a project like this than it is with smaller scale projects, due to the comparatively large amount of work involved, from the research phase to the requirements gathering and design, and finally the implementation itself.

8.3 Working with a supervisor

Working with a supervisor was very useful supervisor throughout the project. Support was always available when needed, and by meeting once a week I was able to get a second opinion on my goals for each Sprint, which saved me from going down what could have been dangerously deep rabbit holes a few times. The migration to Vanilla was one such case, where I had been struggling with the React implementation for over two weeks. During a meeting I mentioned the trouble I'd been having and showed my supervisor the code. As a lecturer with a lot of JavaScript knowledge, he was able to tell me how I was bypassing most of React's functionalities with my attempts to solve the issues at hand, which was making React fairly redundant overall. A suggestion was made to attempt using Vanilla instead, which he acknowledged was a hard pill to swallow, but recommended it all the same. After taking his advice and migrating, I realised that he was entirely right, saving me from a lot more wasted time and effort. Overall, it was a great experience working with my supervisor, and I'm very appreciative of the help I received along the way.

8.4 Technical skills

From a technical perspective I have learnt a lot throughout this project. Besides becoming more familiar with the process of completing a large-scale software project, I've become a lot more comfortable with Vanilla JavaScript, which I had only used a few times throughout my studies, as frameworks such as Vue and React were more of a focus in any web development classes I took. This will help greatly with my confidence around all things JavaScript in the future, as it provides a solid foundational understanding of what is often abstracted away by frameworks and libraries.

I've also gathered a deeper understanding of the Web Audio API and how to work with audio on the web, from using Tone.js to build the inner workings of the synthesizer, to using the extendable-media-recorder library to provide lossless audio recording in the application using the raw audio stream. On the same note (no pun intended), the process of submitting an issue to the Extendable Media Recorder's GitHub repository and communicating with the developer of it, helping him find a bug in his code which solved an issue for us both, has strengthened my appreciation of open source communities, as I now have first-hand experience with the win/win of contributing to open source projects, even if my contribution was just by explaining the problem I had in detail.

8.5 Further competencies and skills

Regarding further competencies and skills that could help me in the workplace, I feel that everything I've learnt throughout will help me in some way or another. As someone who aims to become a frontend web developer, gaining a deeper understanding of JavaScript and frontend development should serve me well in years to come. While learning more about web-based audio programming is a bit more niche, as someone with a huge passion for music, it should help if I ever get my dream job with an audio or music and web-based tech company, such as Bandcamp or Spotify. Lastly, the project management skills I have gained along the way will help me in any job, as well as outside of the workplace, as it has taught me how to manage my time much more effectively.

8.6 Potential further developments

There is no shortage of further developments that could be made to MS24.

To begin with, many more options for sound design could be added to MS24, through extended modulation support, the addition of new effects, and the ability to swap oscillator models (such as FM and AM oscillators) out with different ones. Tone.js could be used to implement a lot of this, but with a bit of work and research, the Web Audio API could be used to create my own custom effects and synthesizer modules.

Additionally, MS24 could be extended from being an almost completely offline app, capable of being loaded with no access to the internet, to a full-stack web app with a server and database. This would allow the creation of a user account system, in which users could register an account on MS24, create their own personal configurations and save the presets to their account rather than just locally. This would let them log in from anywhere and have all the presets they had ever saved available to them. On top of this, a backend would allow for a community presets page, giving users the option to make their synth configurations available to all users. This community aspect would no doubt improve the experience overall, especially for beginner users, who are otherwise restricted to the configurations I've provided as defaults.

To do this, I would likely use MongoDB as the database and the Express.js library running on Node.js as the server. This is mostly because I have plenty of experience with these technologies, and because they are quite simple to use. MongoDB is extremely straightforward, just requiring a free cluster and collection to be created with my account, which would include two main objects: "users" and "community". Users would include each user account, with their username, encrypted password, email, and private presets stored. Community would include an array of presets, which would fill up as users share their configurations publicly. An admin system might be necessary if community presets were to be a thing, however, as it would be difficult to ensure that users didn't use offensive language in their presets' metadata. There might be ways around that though, such as by using AI-powered tools to check for offensive language and URLs, or by having a strong validation system using a database of offensive language to compare against. The server-side codebase would be quite straightforward too, especially if the need for admin accounts wasn't deemed necessary.

This process was actually started towards the end of the implementation phase, just before the major keyboard logic and GUI reworks. However, after creating the MongoDB cluster, as well as installing the required libraries and creating the necessary folder structure for the server, I decided that I would go back to working on the frontend, as I felt I should prioritise the major bugs over user account functionality. It seemed to me that it would be better to provide a solid, mostly bug-free offline synthesizer experience rather than a bug-filled online one. On the following page I have included a screenshot of this project structure and Users collection on MongoDB.

Overall, there are many ways MS24 could be further developed, and there's a high chance I will develop it further in my own time, as I'm genuinely interested in how far MS24 could go, given enough time and effort.

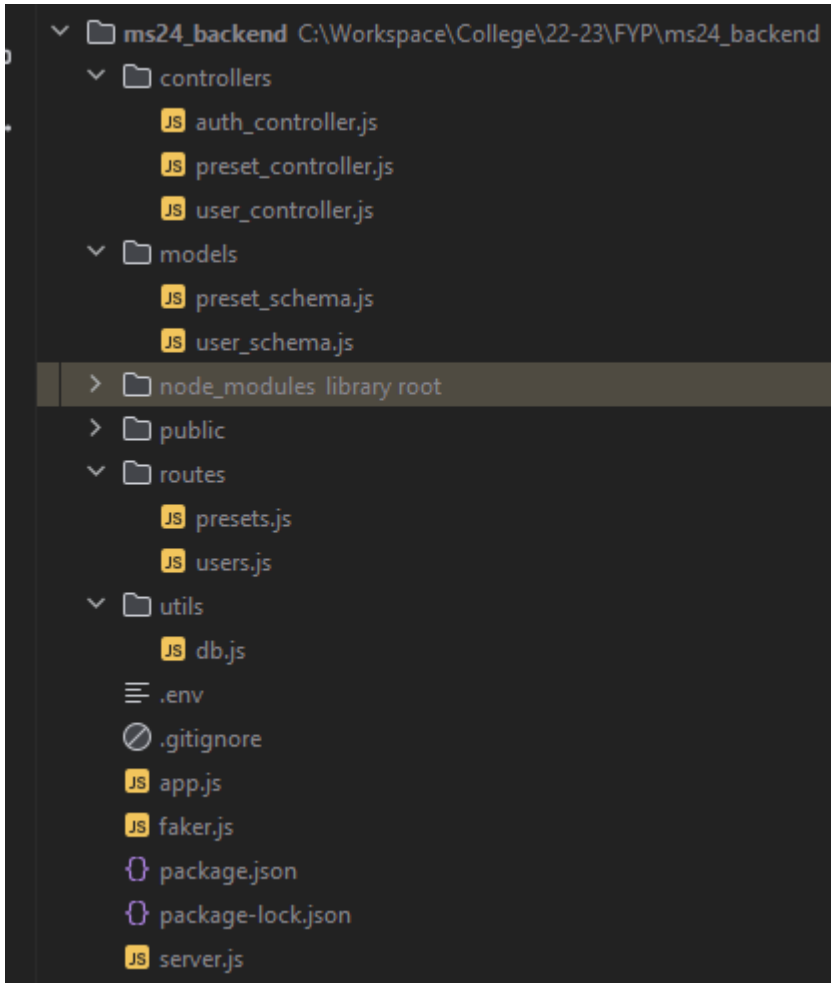


Figure 105. The project structure created for the MS24 server (using Node.js and Express.js)

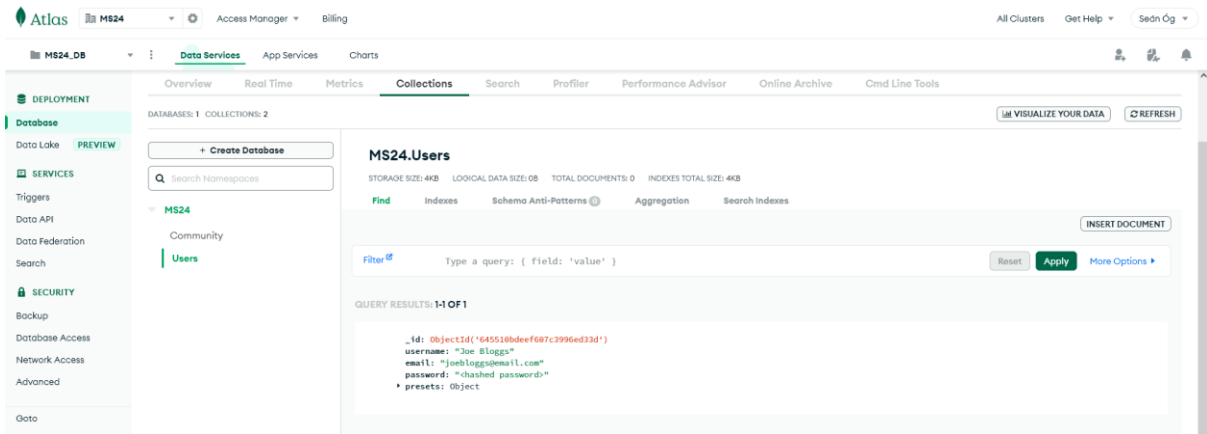


Figure 106. MongoDB MS24 database (Users collection)

9 Conclusion

In conclusion, the development of MS24 aimed to determine the feasibility of the web browser as a platform for music production and audio synthesis. The successful implementation of the application provided an experience as close as possible to a traditional desktop-based software synthesizer, enabling the creation of unique and interesting sounds without any of the prerequisites typically needed to do so. The successful completion of this project demonstrated the potential of the web browser as a platform for music production.

The combination of technologies used in the development of the application provided a powerful platform for audio synthesis and web application development. The use of JavaScript, HTML, TailwindCSS, webaudio-controls, Tone.js, and Vite resulted in an enjoyable and user-friendly application. The successful integration of these technologies demonstrated their suitability for web-based audio synthesis and provided a foundation for future developments in this area.

The research phase of the project was conducted at the beginning, used to inform all future decisions over the design and implementation of the application. The research process consisted of examining the properties of sound, digitally created waveforms, and a literature review examining past research on the Web Audio API, with special focus on the feasibility of using the interface to create web-based music production tools.

The requirements phase of the project was conducted after the research and consisted of gathering requirements for the application through the examination of similar applications and by launching a survey focused on discovering what synthesizer users found to be the most important aspects of audio synthesis, and what non-synthesizer users thought of various synthesizer's user interfaces.

The design phase used the findings from the requirements phase to lay out a plan for the implementation of the project, by firstly creating the program design, focused on the overall technologies used and how they interconnected, followed by the user interface design which focused on how the application would look.

The implementation phase of the project involved building the application by using the plans set out in the design phase over the course of eight Sprints, each consisting of two weeks with the last Sprint as an exception to that rule, consisting of one week. Development was primarily conducted with WebStorm, a web-focused IDE from JetBrains, with constant testing throughout by making use of its debugging capabilities and smart code interpretation.

The testing phase sought to examine how well the implementation met the requirements of the project. This was done through functional testing and usability testing. Functional testing was conducted manually by comparing the expected outputs of inputs (determined by the functional requirements) to the actual outputs. Usability testing was conducted by running five in-person tests on participants to gather information on how usable the application was for users of various technical aptitudes.

Over the course of this project, the author has developed their technical competencies and gained practical experience in several areas, including web development, and audio programming.

MS24 has a great potential for future development. Not only are there many extra features that could be added, but the entire app could be transformed into a full-stack application, complete with a server and database, to offer a user account system and a community presets page.

10 Bibliography

Drumond, C. (n.d.). *Scrum - what it is, how it works, and why it's awesome*. Atlassian.

Retrieved May 1, 2023, from <https://www.atlassian.com/agile/scrum>

Farnell, A. (2010). Chapter 6. Psychoacoustics. In *Designing Sound* (p. 78). essay, MIT Press.

MozDevNet. (n.d.). *Autoplay guide for media and web audio apis - web media technologies:*

MDN. Web media technologies | MDN. Retrieved February 10, 2023, from

https://developer.mozilla.org/en-US/docs/Web/Media/Autoplay_guide

Snoman, R. (2019). *Dance Music Manual: Tools, Toys, and Techniques* (Fourth). Routledge.

State of JavaScript 2022. Front-end Frameworks. (n.d.). <https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/>

11 Annotated Bibliography (Literature Review)

Adenot, P., & Choi, H. (Eds.). (2021, June 17). *Web audio API*. W3C. Retrieved January 3, 2023, from <https://www.w3.org/TR/2021/REC-webaudio-20210617/>

This World Wide Web Consortium (W3C) specification edited by Adenot et al. (2021) and produced by the W3C Audio Working Group presents the Web Audio API as an official “W3C Recommendation”, a type of specification which states the endorsement of a web technology as a standard by W3C members and the director. This specification describes the high-level Application Programming Interface (API) used for processing and synthesizing audio in web applications, and officially recommends it after 11 previous editions of the specification since 2011.

Buffa, M., Lebrun, J., Kleimola, J., larkin, O., & Letz, S. (2018). Towards an open web audio plugin standard. *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, 759–766. <https://doi.org/10.1145/3184558.3188737>

Buffa et al. (2018) put forward their proposal for an open Web Audio plugin standard, mimicking the native plugins often found in Digital Audio Workstations (DAWs). The paper contains work originally being done separately by three groups of researchers with different initial interests. Due to the similarities between their work, these groups decided to work together in developing interoperable Web Audio plugins and plugin hosts, and to focus their efforts “towards the beginnings of an open standard”. One group had been developing the FAUST domain specific language (DSL) since 2002, which could be compiled to several targets including Web Audio, while another group had been developing the Web Audio Module (WAM) since 2014, which use a C++ API, the same language as used to develop almost all native plugins. Together these groups work towards a converged API, hoping to submit their findings to the W3C Audio Working Group for evaluation. This paper contributes important data about the possibilities of bringing more systems and concepts found in DAWs into the Web Audio realm.

Choi, H. (2018). *AudioWorklet: The future of web audio*. hoch.io. Retrieved December 27, 2022, from <https://hoch.io/media/icmc-2018-choi-audioworklet.pdf>

Choi (2018) presents the *AudioWorklet* interface, a highly anticipated Web Audio API enhancement first introduced in a draft specification in 2014, but not released to the public until 2018. This new interface replaced the *ScriptProcessorNode*, an early solution to high demand for more extensibility and flexibility within the Web Audio API which failed to meet the expectations of many developers. This article presents a “comprehensive comparison between AudioWorklet and its predecessor”, providing examples of usage, discussing the technical aspects of the new interface, and highlighting what this new functionality offers the computer music community.

Eriksson, O. (2013, June 25). *Implementing virtual analog synthesizers with the web audio API: An evaluation of the web audio API*. DIVA. Retrieved January 5, 2023, from <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A632732&dswid=-1886>

Eriksson (2013) presents his inductive thesis, in which the implementation of “virtual analog synthesizers” with the Web Audio API is evaluated. After a reference architecture is defined from six existing synthesizers, Eriksson develops a categorization and scoring system to determine the viability of modelling a synthesizer with the API. By using this system, the Web Audio API receives a score of approximately 80%. Eriksson concludes that the main thing the API lacks is flexible modulation across each of its nodes. This paper provides useful insight into the capabilities of the Web Audio API regarding modelling analogue synthesizers.

Jensenijs, A. R., Lyons, M. J., Roberts, C., Wakefield, G., & Wright, M. (2017). 2013: The Web Browser as Synthesizer and Interface. In *A NIME Reader: Fifteen Years of New Interfaces for Musical Expression* (Vol. 3, pp. 433–450). essay, Springer International Publishing.

Jensenijs et al. (2017) introduce the Web Browser as one of the thirty most influential New Interfaces for Musical Expression (NIMEs) from the NIME archive. To fairly select 30 out of over 1200 papers in the NIME archive, the authors initially created a list of around 50 articles which were perceived by the community as “influential”. To narrow it down to 30, to include approximately two published items for every year of the NIME conference, they identified the most cited of these papers using the Google Scholar index. In this chapter, web technologies including the Web Audio API are discussed in relation to NIMEs, from problems and possibilities of the API to the *Gibber* live coding environment and *Gibberish* library. The chapter closes with a section titled “*Expert Commentary: The potential synthesizer in your pocket*” written by Abram Hindle, in which he states the enormous potential of the Web Audio API. The inclusion of the Web Browser and Web Audio API in this anthology from over 14 years of conferences is very encouraging to anyone loPASSEdING to study or develop in this area.

Lind, F., & MacPherson, A. (2017). *Soundtrap: A Collaborative Music Studio with web audio*. webaudioconf. Retrieved January 5, 2023, from https://webaudioconf.com/posts/2017_EA_29/

In this article written for the Web Audio Conference from Lind and MacPherson (2017), Spotify’s collaborative DAW *Soundtrap* is presented alongside a summary of its architecture. It’s ability to connect users from around the world to compose music together while communicating across video chat is described, as well as the web application’s ability to run on a wide range of devices. Some of the challenges being tackled by the Soundtrap development team are also mentioned, such as audio latency and CPU usage. New projects being worked on by the team such as “live jamming” are also described. This article serves as an example of one of the many web-based digital audio workstations built using the Web Audio API.

Mann, Y. (2015). *Ressources.ircam*. Interactive Music with Tone.js - Ressources. Retrieved December 26, 2022, from <https://medias.ircam.fr/x9d4352>

Mann (2015) describes Tone.js, a Web Audio framework which facilitates the creation of interactive music in the browser through a myriad of premade classes, each representing objects found throughout computer music software. The similarity of Tone’s building blocks

to modules and systems originating from DAWs, such as the built-in transport system and audio buses is explored by Mann, before comparing Tone to frameworks with similar functionality. The paper concludes by looking at possible future work on Tone.js, mentioning some of the hurdles in developing music for the browser, before finally envisioning the creation of interactive, collaborative musical spaces with the addition of WebSockets or WebRTC.

Roberts, C., & Kuchera-Morin, J. A. (2012). *Gibber: Live coding audio in the browser*. ResearchGate. Retrieved December 25, 2022, from https://www.researchgate.net/publication/283595161_Gibber_Live_coding_audio_in_the_browser

Roberts et al. (2012) present *Gibber*, a live coding environment for web browsers focused on audio synthesis and sequencing, written in pure JavaScript. With an array of synthesis options available, and the ability to run “networked performances” in which multiple users remotely & simultaneously control an instance of *Gibber*, it is shown that a real-time collaborative jamming space can be provided through the web browser. The paper closes with Roberts et al. stating their excitement at the educational potential of such an environment, before describing some of the future work required to develop *Gibber* further.

Roberts, C., Wakefield, G., Wright, M., & Kuchera-Morin, J. A. (2015). Designing musical instruments for the browser. *Computer Music Journal*, 39(1), 27–40. https://doi.org/10.1162/comj_a_00283

This journal from Roberts et al. (2015) explores the great potential of web technologies for musical expression. The paper introduces two JavaScript libraries: *Gibberish.js* and *Interface.js*, before showcasing a complete system working in both desktop and mobile browsers, which incorporates these libraries into the live coding environment *Gibber*. The problems and possibilities of the Web Audio API are then explored, in which the shortcomings of the *ScriptProcessorNode* are explained, as well as the *AudioContext* upgrade that was on the way at the time of writing. Following this, Roberts et al. describe the process of instrument design within *Gibber*, by going through the various mappings between audio, visual, and interactive elements such as a slider used for picking a frequency. The paper concludes by giving several examples of educational institutions using these technologies and provides a summary of recent tests and future development plans.

Roberts, C., Wright, M., Kuchera-Morin, J., & Höllerer, T. (2014). Rapid Creation and Publication of Digital Musical Instruments. In *NIME* (pp. 239-242).

In this NIME conference paper, Roberts et al. describes “research enabling the rapid creation of digital musical instruments and their publication to the internet” and examines the cross-compatibility of web-based instruments, as well as the attractiveness of the web browser as a platform for digital musical instruments. The paper also explores the simplification of creation of works, and the increased accessibility to both the developer and end-user granted by the medium. This paper acknowledges the incredible opportunities presented by the browser “for authoring and disseminating digital musical instruments”.

Smus, B. (2013). A Brief History of Audio on the Web. In *Web Audio API* (pp. 1–11). essay, O'Reilly.

In this chapter of the *Web Audio API* book from O'Reilly, Smus (2013) gives historical context by exploring the history of audio on the web, from the earliest HTML implementation, the `<bgsound>` element, to the HTML5 `<audio>` element, to the JavaScript based Web Audio API. Smus explores the usage of audio on the web through games, user interfaces and interactivity before providing detailed explanations of the API's audio context, including diagrams to help understand more complex configurations. The chapter then goes on to present some critical sound theory, in both the physical and digital realms, before describing the various nodes made available through the Web Audio API. The chapter closes by showcasing basic Web Audio JavaScript code and provides instructions on how to get started. Understanding what came before the Web Audio API and the limitations of the earliest implementations of audio on the web is useful in seeing how far we've come from simple sound effects played on mostly static web pages.

Web Audio API brings audio design to the Web as it becomes a World Wide Web Consortium (W3C) Recommendation. (2021, June 17). W3. Retrieved December 20, 2022, from <https://www.w3.org/2021/06/pressrelease-webaudio.html.en>.

In this press release by the World Wide Web Consortium (W3C) in 2021, the Web Audio API is officially presented as a W3C Recommendation. The API's functionalities and wide range of applications are described, with W3C stating that the standardization of the API has led to it becoming "a dependable, widely deployed" suite of tools, with several examples of Web Audio usages such as SoundCloud, Ableton, and Spotify being given. The new possibilities in contrast to most native audio applications are explored, and the release informs of the W3C Audio Working Group already working towards the Web Audio API v2, with plans to "enrich the first version of the API" with many complex and highly requested features "which were insufficiently developed to be included in the first version of the API". Finally, the release gives testimonials from W3C members and Industry Users, including the BBC, Google, Mozilla, and Spotify. This press release serves as an example of the importance of the Web Audio API and proof of its improvement over the years, allowing it to eventually become a standard on the Web.

Wyse, L., & Subramanian, S. (2013). The viability of the web browser as a computer music platform. *Computer Music Journal*, 37(4), 10–23. https://doi.org/10.1162/comj_a_00213

Wyse et al. (2013) explore the historical boundary-pushing of technologies in the computer music community, giving examples of many systems and protocols that were developed to serve the community. The Web Audio API standard and related technologies are discussed in detail, with special attention given to the capabilities of these systems and integration of them in the web browser. The suitability of the web browser is examined, especially regarding "critical aspects of audio synthesis, timing, I/O, and communication". The researchers compare the Web Audio API to the native alternatives and attempt to understand how well it can meet the needs of the community, identifying timing and extensibility as two key areas that still need work. Through this extensive research by Wyse

et al. it is clear that the feasibility of the web browser as a platform for music production and sound design is on the rise, especially due to the standardization of the Web Audio API and through the optimisation of JavaScript over the years, closing the gap between the browser and native platforms.

12 Appendices

Survey

<https://forms.gle/L7gK8KfHG6WeAe138>

Figma designs

[https://www.figma.com/file/0bTQPnDboqvkfFb4gLqtuO/FruitySynth-Wireframes-V2-\(Copy\)?node-id=604%3A2&t=sSEul4JdhYVIQGwN-1](https://www.figma.com/file/0bTQPnDboqvkfFb4gLqtuO/FruitySynth-Wireframes-V2-(Copy)?node-id=604%3A2&t=sSEul4JdhYVIQGwN-1)

GitHub repo

<https://github.com/IADT-projects/y4-project-Mangoshi>