Forma Sonus:
A Browser-Based Audio-Reactive Visual System

Jack Lalor

N00222381

Supervisor: Michael McAndrew

Second Supervisor: John Montayne

Year 4

DL836 BSc (Hons) in Creative Computing

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 to 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 : Jack Lalor

Signed

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

Contents

Abstract

Forma Sonus is a browser-based application designed to create audio reactive visuals in real time. The project focuses on making visual systems more accessible, especially for users without programming experience or experience with complex visual software. The application lets users input audio, design and customise pre-built base visual types, and combine them to present for viewing.

The result is an interactive tool that enables users to experiment with audio reactive visuals in real time. Overall, the project shows how real-time audio visualisation can be simplified while still allowing users creative freedom.

Acknowledgements

Firstly, special thanks go to the project supervisor for their guidance and feedback throughout the past several months of this project's development. From taking time out of their day to schedule extra meetings when it was needed most, to giving very honest and detailed input which helped guide the project to what it is today, thank you.

Acknowledgement is also owed towards the second supervisor who gave honest feedback, which ultimately led towards reframing the project from TouchDesigner-based to a web-based project. Thanks, are also given towards all of the lecturers over the past few years of the creative computing course for their support and the knowledge they provided.

Lastly, thanks go to the individuals who assisted with testing and provided feedback during development; their input after testing greatly improved the project.

1. Introduction and Project Context

1.1 Introduction

Visuals are often used alongside music to add more interest for the audience (Bain, 2008). This is seen in many different contexts, from live gigs and DJ sets to digital art and even social media videos. When visuals react to the music, such as changing colour or moving in time with the beat or specific frequencies, it makes the whole experience feel more connected and immersive for the crowd or viewers. That is why DJs, musicians, and content creators love using audio reactive visuals to enhance their performances or make their content stand out.

One big issue is that most of the tools out there for making these visuals are complicated. They often require the user to learn specialised software, experiment with node graphs, or even code. For someone who is new to this field, it can be difficult to get their head around how some of these complex technologies work, and it is easy to get stuck before the user can even make anything substantial.

Forma Sonus was created to make audio-reactive visual creation easier to understand and more accessible to users. Instead of making people build everything from scratch, the app provides a set

of base visuals the user can experiment with and customise using simple sliders and controls. Users can see straight away how changing a setting affects the way the visuals react to sound, without needing to know any programming or advanced software. The idea behind Forma Sonus is to help people get a feel for how things like frequency, amplitude, and motion can change what appears on screen, and to make it less intimidating for beginners to start experimenting with audio-reactive visuals.

The name Forma Sonus is derived from Latin, where forma refers to shape and sonus refers to sound, so the English translation of this project is “Shape of Sound”. The name reflects the core idea of the project, which is to transform sound into visual form.

The application uses common web technologies, enabling it to operate directly within the browser without any need for software installation. Visuals can animate on their own or react to audio input, giving users options for testing their audio against visual elements. By balancing simplicity with creative control, the project strives to make visualisation concepts more approachable for newcomers to the field.

1.2 Project Background

Audio visualisation has been used for many years as a way of representing sound through visual elements such as shapes, colour and movement. Early examples of audio visualisation were shown in media players such as Winamp (Nullsoft, 1997), where graphical elements responded to music with frequency bars and waveform displays (Foote, 2000). Over time, audio-visualisation tools have developed into more advanced systems capable of generating complex animations that respond to different aspects of sound. (Bain, 2008; Graf et al., 2021)

Modern creative software allows designers and developers to build interactive visuals that respond to live audio input. Tools such as TouchDesigner (Derivative, 2020) and other creative coding frameworks allow users to create real-time graphics that can be used in live performances, digital installations and multimedia projects. These tools often provide a high level of control, allowing visuals to be influenced by audio features such as frequency band or amplitude. However, this software can also require a very long time to learn, especially for users who are new to visual design and programming.

Earlier versions of the project looked at using TouchDesigner to generate audio-reactive visuals, including the idea for using real-time AI-generated imagery. However, this direction was left behind for multiple reasons, which will be explained in further sections, ultimately leading to a shift in focus toward creating a browser-based application that gave similar interactive visuals more efficiently and accessibly.

As web technologies have improved, it has become possible to create real-time graphics directly in the browser using JavaScript graphics libraries such as Three.js (Cabello, 2010). Three.js provides an easier way to work with 3D graphics by building on top of WebGL, allowing developers to create interactive visual experiences without needing to work directly with low-level graphics code. Libraries such as React Three Fiber (Poimandres, 2019) further simplify this process by allowing Three.js to be used within React applications, making it easier to combine interactive visuals with modern user interfaces. Browser based tools also allow applications to be shared easily and accessed across different devices without requiring installation of specialised software. (Mozilla Developer Network, 2023)

Overall, the project background highlights how audio visualisation has evolved from simple visual effects to more complex real-time systems, and how modern web technologies make it possible to create accessible tools for exploring these ideas.

1.3 Problem Statement

Many existing tools for creating audio-reactive visuals are designed for experienced users and often involve complex workflows or specialised software. (Bain, 2008) Although these tools offer a lot of control, they can be difficult for beginners to get used to and often take time to create something they are happy with. This can make it harder for new users to understand how audio affects visual behaviour.

More straightforward visual tools exist, but they often limit customisation options, restricting experimentation. This creates a gap between ease of use and creative freedom, making it difficult for beginners to find tools that allow them to discover audio-reactive visualisation in an understandable way.

Forma Sonus aims to address this problem by providing a browser-based tool that introduces the core concepts behind audio-reactive visuals in a more approachable, interactive way.

1.4 Aims and objectives of the Project

The aim of this project is to design and develop an interactive web-based application that allows users to create real-time visuals that can optionally react to audio input. Forma Sonus was designed to offer users an easier way to understand the fundamental concepts of audio-reactive visualisation, without needing advanced technical skills or familiarity with complex visual software. The project focuses on providing a set of base visual systems that can be customised via parameter controls, allowing users to experiment with how visual properties such as movement, colour and scale are influenced by sound.

The project's objectives were defined to guide the development process and ensure the application meets its intended goals.

The main objectives are:

- To research existing audio visualisation tools and identify common visual techniques used to represent sound.
- To design a browser-based system capable of generating real-time visuals.
- To implement audio analysis that allows visual properties to respond to sound input.
- To develop a collection of predefined visual modules that can be customised by the user.
- To create a layer system that allows multiple visuals to be combined together.
- To design an interface that allows users to easily adjust visual parameters.
- To evaluate the performance and usability of the application.

These objectives aim to create a system that educates users on the core concepts of audio reactive visuals while maintaining room for creative expression.

1.5 Target Users

Forma Sonus is aimed at users who are interested in creating visuals that can be used alongside music or as standalone visual displays. One of the main target users includes musicians and DJs/VJs who may want visuals that react to audio during live performances or recorded mixes.

Content creators and streamers are also considered target users, as animated visuals can be used as backgrounds for livestreams or video content. Using real-time visuals can help make content more visually interesting without requiring advanced video editing skills.

It could also be useful for creatives working on visuals for art installations, exhibitions or festival environments. Visuals are often used in these types of settings to help create atmosphere or add another layer to the overall experience. Since the visuals can either react to sound or run independently, the application can be used in situations where audio may or may not be present. This makes the tool suitable not only for music-related use but also for creative projects in which visuals are part of a broader artistic setup.

Overall, the application is designed for users who want an easier introduction to audio reactive visuals while still allowing room for creative experimentation.

1.6 Scope of the Project

The scope of this project is to design and develop a browser-based application that allows users to create and customise real-time visuals, optionally reacting to audio input. The project includes a set of predefined visual types which can be adjusted using parameter controls such as colour, movement, scale and sensitivity to audio. Users are able to combine multiple visuals together using a layer system, allowing more complex visual outputs to be created.

The application supports different audio input methods, including audio file upload, microphone input and line-in input from an external audio device. A full-screen Present Mode is also included so that visuals can be displayed without the UI being displayed, making the application suitable for live performances, digital displays or creative projects.

The project focuses on providing an accessible introduction to audio-reactive visualisation rather than building a fully professional visual production tool. Features such as advanced video exporting, complex node-based editing systems or collaborative online functionality are outside the scope of this project. The goal is to create a system that demonstrates the core concepts of audio reactive visuals while remaining easy to use and understand.

1.7 Success Criteria

The success of Forma Sonus is measured by how easily users are able to create and customise real-time visuals. Users should be able to adjust visual parameters and immediately see the changes reflected in the output. Visuals should run smoothly while the application is in use, without noticeable performance issues when using normal settings.

Another measure of success is whether users are able to understand how visual elements respond to audio input. The application should clearly demonstrate how properties such as frequency and

amplitude can influence movement, scale and colour. This supports the goal of helping users learn the core ideas behind audio reactive visualisation.

The interface should be simple to use, so users can experiment with various visual styles without requiring technical expertise. The layer system needs to support combining multiple visuals seamlessly, ensuring the application remains stable.

Overall, the project can be considered successful if users are able to create visuals that respond clearly to parameter changes while also developing an understanding of how audio can influence visual behaviour.

2. Research and Background

2.1 Development of Audio Reactive Visuals

Audio reactive visuals have been used for many years as a way of representing sound through visual elements such as shape, colour and movement (Bain, 2008). Early media players, including Winamp, introduced basic forms of audio visualisation, where simple graphics responded to sound in real time. These early visualisers helped introduce the concept of linking sound with visual feedback in real time.

As technology developed, audio visualisation became more advanced and started to appear in live performances, digital art installations and multimedia experiences. VJ-ing (Video Jockeying) became popular as artists began mixing visuals live alongside music performances, using software tools to manipulate video, graphics and effects in real time. Research has shown that combining music with responsive visuals can improve audience immersion and create a stronger connection between sound and image (Fujishiro et al., 2018; Bain, 2008).

Modern audio visual systems allow visuals to respond dynamically to sound using techniques such as frequency analysis, amplitude tracking and beat detection. These developments have allowed artists and developers to create more immersive and interactive experiences where visuals behave in a way that reflects the structure and energy of music.

2.2 Audio Features Used in Visual Systems

Audio features play an important role in determining how visuals respond to sound. Different characteristics of audio can be analysed and used to control visual behaviour in real time. Common audio features used in audio-reactive systems include amplitude, frequency bands and beat detection (Graf et al., 2021).

Amplitude represents the overall loudness of the audio signal and is commonly used to control large visual changes like scaling, brightness or intensity (Graf et al., 2021). Louder sections of audio can result in larger or more noticeable visual movement, while quieter sections may produce more subtle changes.

More detailed visual responses can be taken by analysing how audio energy is distributed across different frequencies. This allows visuals to respond differently depending on whether the sound contains low, mid or high frequencies.

2.2.1 Frequency Analysis

Frequency analysis is used to understand how sound energy is distributed across different parts of the frequency spectrum (Lyons, 2010). Instead of treating audio as a single value, frequency analysis separates the signal into multiple components, allowing visuals to respond to specific characteristics of the sound.

For example, bass sounds such as drums or kick patterns occur in lower frequency ranges, while instruments such as vocals or synths may occupy mid or higher frequency ranges. By analysing these different areas of the sound spectrum, visual systems can produce more varied and detailed responses.

Frequency analysis allows different parts of a visual to react independently, creating more complex and dynamic behaviour compared to using amplitude alone.

2.2.2 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is one of the most commonly used techniques for analysing audio frequencies in real time (Lyons, 2010). FFT works by converting audio signals from the time domain into the frequency domain, allowing the system to measure how much energy is present at different frequency ranges.

FFT makes it possible to break audio signals into smaller components, which can then be mapped to visual properties. Many real-time audiovisual systems rely on FFT because it provides a fast and efficient way to analyse sound continuously while the application is running (Graf et al., 2021).

Using an FFT enables visual elements to respond more precisely to sound features, making the visuals seem more integrated with the audio structure. (Lyons, 2010).

2.2.3 Frequency Bands (Low, Mid, High)

Frequency data taken from FFT is grouped into three as low, mid and high frequencies. These bands allow visual behaviour to be linked to different parts of the audio spectrum.

Low frequencies typically include bass sounds such as kick drums or basslines, which are often used to control large visual movements such as scaling or pulsing effects. Mid frequencies may influence elements such as shape deformation or motion patterns, while higher frequencies can be used to control smaller details such as colour changes or visual intensity.

Separating audio into frequency bands allows visuals to work in a more structured way, where different elements respond to different characteristics of the sound. This helps create more visually interesting results and avoids visuals reacting too uniformly to all audio input. (Graf et al., 2021)

2.3 Mapping Audio Data to Visual Parameters

Mapping audio data to visual parameters is an important process in audio-reactive systems, as it determines how sound influences visual behaviour. Audio features such as amplitude and frequency values can be connected to visual properties, including scale, colour, and rotation**.** The way audio data is mapped has a strong influence on how the visuals appear and how closely they reflect the structure of the sound.

If audio values are used directly without any adjustment, the visuals can end up reacting too strongly or behaving in an unstable way. Small changes in the sound can cause sudden or abrupt movement, making the visuals harder to follow. Because of this, techniques such as scaling, smoothing and sensitivity adjustments are applied to make the behaviour more controlled (Graf et al., 2021). These changes help keep the motion more consistent and easier to read.

2.4 Real-Time Graphics in the Browser

Real-time visuals need to update continuously while the application is running, especially when they are reacting to audio input (Graf et al., 2021). For visuals to feel connected to sound, changes need to happen instantly, without noticeable delay. Modern web technologies now make it more possible to create real-time visuals directly in the browser, which allows a user to create and run interactive visuals without needing specialised software. This makes visual tools more accessible as users can open the application on most devices without needing to install additional programs.

2.4.1 WebGL

WebGL (Web Graphics Library) *(Khronos Group, 2011)* is a javascript API which allows for 2D and 3D graphics to be rendered using the computer's graphics processing unit (GPU). Using the GPU allows for the visuals to be displayed more efficiently compared to relying only on the computer's main processor. This makes it possible to render animations and visual effects smoothly, even when many visual elements are present on screen at the same time.

WebGL is one of the main technologies used by many browser based graphics libraries. It enables developers to create interactive visuals that can be continuously updated, which is crucial for applications dependent on real-time feedback.

2.4.2 Three.js

Three.js is a JavaScript library that makes it easier to create 3D graphics in the browser, by providing built in functions for creating shapes, lighting and animation, instead of requiring developers to write graphics code. It is built on top of WebGL and provides simpler ways to create geometry, lighting, materials and animations. Instead of working directly with complex graphics code, developers can use Three.js to focus more on how visuals behave and interact.

Three.js is widely used in creative coding because it allows interactive visual projects to be created more easily while still maintaining good performance. It allows visual elements such as particles, shapes and animated objects to be rendered in real time. This then makes it suitable for audio reactive visuals, where graphics need to respond quickly to changes in sound.

2.4.3 React Three Fiber

React Three Fiber allows Three.js to be used within a React application (Poimandres, 2019). It allows 3D visuals to be structured using React components, making it easier to organise visual elements alongside user interface elements such as sliders or buttons. Using React Three Fiber makes it easier to manage visual components as part of the overall application structure. Because Forma Sonus includes multiple visual types and adjustable parameters, this approach helps keep the code organised and easier to manage. It also allows visual elements to update efficiently when parameters change, helping maintain smooth real-time behaviour.

2.5 Existing Tools and Applications

There are many tools available for creating audio-reactive visuals, ranging from simple music visualisers to more advanced real-time visual software. Creative coding frameworks such as Processing and p5.js (McCarthy et al., 2015) allow developers to create generative visuals through programming, while professional tools are commonly used in live performances, installations and multimedia projects. Most of these tools allow visuals to respond directly to audio input, causing movement or visual changes based on the music.

TouchDesigner (Derivative, 2023) is one of the most popular tools for creating up to industry standard real time visuals, particularly in live performance environments. It uses a node based workflow which allows different elements of a visual system to be connected together. Audio data can be analysed and linked to visual parameters, allowing visuals to respond to sound in real time. While TouchDesigner offers a high ceiling for possibility it can also take a lot of time to learn due to the complexity of node based workflows, especially for a person not familiar with programming. This can make it more challenging for beginners to quickly test out audio reactive visuals.

Another example of tools used for creating generative visuals which was mentioned previously includes frameworks like Processing and p5.js. These tools allow visuals to be created through code, including visuals that can react to audio. They are often used in education and digital art because they help users learn how visual behaviour can be controlled through programming. However, they still require the user to write code in order to produce results, which can be difficult for someone who is new to programming. While these frameworks offer a lot of flexibility, they do not always provide a quick way for beginners to test ideas or experiment with audio-reactive visuals.

Looking at these tools shows that there are different ways to create audio reactive visuals, but they are not all equally easy to use. Some tools offer a lot of control but can be difficult to learn, while others are simpler but offer little customisation. Because of this, it can be hard for beginners to find a tool that is both flexible and easy to understand. Forma Sonus draws inspiration from these tools but emphasizes offering core visuals that users can tweak via parameter controls. This enables users to experiment with visual behaviour and understand how audio impacts visuals.

2.6 Research Summary

This chapter's research enhanced understanding of audio-reactive visual systems and how sound controls various visual behaviours. It explained concepts like amplitude, frequency analysis, and FFT, illustrating how audio data can influence visual aspects such as movement, scale, and colour. Additionally, the research emphasised the necessity of real-time rendering, as visuals must update continuously to stay synchronised with the audio.

Looking at existing tools showed that there are different ways to create audio reactive visuals, but they also vary in how easy they are to use. Some tools require programming knowledge or experience with complex software, making it harder for beginners to experiment with ideas. This influenced the decision to design Forma Sonus as a browser-based application that uses predefined visual types, which can be adjusted using parameter controls.

Overall, the research indicates that making audio reactive visuals more accessible can be achieved by simplifying the way sound influences visual behaviour. These findings informed the design choices during the creation of Forma Sonus.

3. Requirements Analysis

3.1 Functional Requirements

Functional requirements of Forma Sonus focus on the main features needed to create real time visuals in the browser. The application lets users design visuals that can either animate by themselves or react to audio input, depending on how the user wants their visuals to run. This allows users to experiment with visual styles without needing an audio source, while still supporting audio reactive behaviour when sound input is available.

One of the core requirements of Forma Sonus is the ability to capture and analyse audio in real time. The application supports multiple audio input methods including audio file upload, microphone and line in input. Having multiple input options allows the application to be used in various situations, whether that's testing visuals on a local device or connecting it to a mixer for live use. The audio is analysed using FFT (Fast Fourier Transform), which breaks the sound into frequency bands (Lyons, 2010) such as lows, mids and highs. These values can then be used to control how visuals behave, for example, affecting movement, scale or colour when audio reactivity is turned on.

Another requirement of the system is the ability to render visuals continuously while the application is running. Visual elements update in real time so that movement appears smooth and responsive. Visuals are also able to animate without audio input using internal parameters such as rotation speed, wave motion and scale changes.

Forma Sonus provides a set of predefined visual modules which act as the base for visual creation. These include Frequency Bars, Particle Wave, Shape Grid, 3D Audio Shapes, Waveform visualisation and Text Visual. Instead of creating visuals from scratch, users customise these base visuals by adjusting parameters that control their appearance and behaviour. Each visual responds differently, allowing users to experiment with different styles depending on what they are trying to create.

The system also includes a layer feature which allows to layer multiple visuals together. Each layer stores its own settings, allowing visuals to be adjusted independently without affecting other layers. This makes it possible to create more complex visual pieces by combining different visual styles.

Users must also be able to adjust visual parameters in real time. Each visual includes controls for properties such as colour, sensitivity, scale, spacing and animation behaviour. Controls are provided through sliders, colour pickers, toggle switches, numeric inputs and button grids, allowing users to experiment and see results immediately.

Forma Sonus also supports saving presets, allowing visual configurations to be reused or prepared before a performance. The application includes both Design Mode for editing individual visuals and Layer Mode for combining multiple visuals. A full screen present Mode is also supported, allowing visuals to be displayed without interface elements when used for projection, livestreams or live performance.

Overall, the functional requirements focus on enabling users to create, customise and combine real-time visuals, with the option to use audio as an input when needed.

3.2 Non-Functional Requirements

As Forma Sonus runs in real time, performance and responsiveness are important factors in making sure visuals appear smooth and react quickly to user input or audio behaviour.

Performance is one of the main considerations, as visuals update continuously while the application is running. The system needs to maintain a stable frame rate so that animation appears fluid and does not feel delayed. Performance can be affected when multiple visual layers are active at the same time, particularly when using more complex visuals such as particle-based types. Maintaining smooth performance is important so that visuals remain visually consistent during creative use or live performance.

Responsiveness is also important, as users should see immediate feedback when adjusting parameters. Changes made using sliders, colour pickers or toggle controls should instantly affect the visual output. When audio-reactive behaviour is enabled, visual changes should appear closely synchronised with the sound so that movement feels connected to the music.

Usability is a key requirement since the application aims to be accessible to users with limited experience in audio visualizer tools. The interface needs to be straightforward so users can quickly start creating visuals. Controls are organised based on the visuals they influence, helping to minimise confusion when adjusting settings. Button grids are used to select visual options, ensuring choices are visible without opening dropdown menus.

Stability is crucial when managing multiple visual layers. Each layer contains its own settings, allowing visuals to be edited without affecting others. The system must remain reliable when switching between visual types or frequently changing parameters.

Compatibility was also a key focus, as Forma Sonus runs directly in the browser and doesn't need extra software installation. It was tested in Google Chrome because of its good support for WebGL and Web Audio features.

Overall, the non-functional requirements focus on making sure the system is responsive, stable and easy to use while supporting real-time visual output.

3.3 Target Users

Forma Sonus is mainly aimed at people who want to experiment with visuals that can be used with music, or simply used as moving visual pieces on their own. The application is intended for users who are curious about how visuals can react in real time, but who may not have experience using complicated visual software or writing code.

One group of users would be musicians or DJs who want something visual to blend with a live set or a recorded mix. Visuals are often used in performances to add something extra alongside the music. They can help create a certain mood or make the performance feel more interesting to watch. Since the visuals in Forma Sonus can react to sound, they change depending on what is happening in the music, for example, when the music becomes louder or more energetic.

Content creators or streamers could also use the visuals as moving backgrounds for videos or livestreams. Instead of creating visuals manually in editing software, the visuals can run on their own and still give the content a more dynamic look.

The application may also be useful for people interested in experimenting with visuals but unsure how to build them from scratch. Forma Sonus provides starting visual types which can then be adjusted using controls such as sliders and colour pickers, allowing users to see how different changes affect the visual behaviour.

Overall, the project is intended for people who want to try working with real-time visuals in a simpler way, especially when combining visuals with sound, while also getting a better idea of how audio can affect how visuals move or change.

3.4 Use Case Scenarios

Forma Sonus can be used in a variety of situations, depending on how the user wants to work with visuals. One common use case is live music performance, where a DJ or musician may want visuals that react to the audio being played. In this situation, the user can connect an audio source, such as a mixer or microphone, so that the visuals respond to changes in the music in real time. The full-screen Present Mode allows the visuals to be displayed on a projector or external screen without showing the application interface, which creates a more immersive viewing experience.

Another use case is creating visual backgrounds for livestreams or recorded video content. Users can design visuals in advance by adjusting parameters and saving presets, then use the visuals as animated backgrounds while streaming or recording. Because visuals can animate without requiring audio input, the application can still be used even when sound is not present.

Forma Sonus can also be used as a tool for experimentation, where users explore different combinations of visuals, colours and motion settings. The layer system allows multiple visuals to be combined together, making it possible to test different visual styles and observe how they interact.

These use cases show how the application can support both real-time performance scenarios and creative experimentation, depending on the needs and preferences of the user.

3.5 Requirements Gathering

Earlier ideas of the project trialled with using TouchDesigner to generate audio-reactive visuals, including the idea of integrating real-time AI-created imagery. However, this approach proved to be too resource intensive for the available hardware and did not fit within the project timeline. The focus moved to developing a browser-based application. This shift also steered the project toward creating a more user-friendly version of an audio-reactive visual tool, allowing users to explore visuals easily without requiring advanced software knowledge or technical skills.

The requirements for Forma Sonus were mainly shaped by looking at existing audio visualisation tools, creative coding frameworks and other software used in live visual performance. After going through these tools, it became clear what features are usually included in audio reactive systems, such as visuals responding to frequency, waveform displays and different types of animated effects.

While researching these tools, it was noticed that a lot of them offer a high level of control, but at the same time they can be quite difficult to learn, especially for someone who is new to this area. Many of them require either programming knowledge or understanding how node-based systems work, which can make it harder to quickly experiment with ideas. Because of this, one of the main things to focus on was making something that felt easier to understand and work with.

Instead of building a system where users had to create everything from scratch, the decision was made to include a set of built-in visual types that could be adjusted using controls like sliders and colour pickers. This approach lets users experiment and see how visuals change quickly, without needing to understand all the technical details behind it.

Another important requirement was making sure everything runs in real time. Audio reactive visuals don't really work properly if there is a delay, so the system needed to update visuals continuously as the audio changes. This influenced a lot of the technical decisions, especially around how audio is analysed and how the visuals are rendered in the browser.

Accessibility was also something kept in mind. Many of the tools require installation or specific setup, which can be a barrier. By making Forma Sonus run in the browser, it removes that step and makes it easier for people to just open it and start using it straight away.

Overall, the requirements came from both researching what already exists and thinking about how to make something that is simpler to use, while still letting people experiment with audio reactive visuals in a meaningful way.

4. Design

4.1 Design Overview

Forma Sonus is a web application developed with React, React Three Fiber, Chakra UI and Three.js. It performs real-time audio analysis via the Web Audio API, visuals are displayed as layers that users can customise and combine. The user interface features controls for parameter adjustments, with all states managed centrally to make sure there is immediate feedback. Users can export and import setups as JSON files.

4.2 System Architecture

Forma Sonus' system architecture is designed around a browser-based framework that manages both audio data and visuals in real time. React (Meta, 2023) is used to manage the interface and the overall application state, while React Three Fiber is used to render the visuals through Three.js. Audio input and processing are handled using the Web Audio API (MDN Web Docs, 2023), which makes it possible to capture and analyse sound in the browser.

The system is split into a few main parts. Audio input can come from different sources such as uploaded audio files, a microphone or an audio device through a line-in input. Once audio is received, it is passed into an analyser where FFT is used to break the signal into frequency values. These values are updated continuously while the application is running.

The processed audio data is then used by the visual components. Each visual uses the data in its own way, depending on how it has been set up or, in some cases, what the user wants the data to target. For example, in the 3D Shapes visual, the user has the choice of what frequency gets to influence

rotation speed and scale pulse. See Appendix E *(Figure E.1)* for an example of the user's option for audio mapping. The visuals are rendered using React Three Fiber, which enables them to update easily when new data is received.

User input is also part of the system. Controls such as sliders, toggles and colour pickers let different visual parameters be changed. These inputs update the application state, which then directly affects how the visuals are displayed. Because of this, changes can be seen in real time, which makes it easier to test and try out different settings.

The layer system enables multiple visuals to be active simultaneously. Each layer retains its own settings, ensuring that changes to one visual do not impact others. This setup allows combining various visual types to produce more complex outputs.

Overall, the system is structured so that both audio data and user input manipulate the visuals at the same time. This helps keep everything connected while keeping the application responsive during use.

4.2.1 Audio Processing Pipeline

The Audio processing pipeline is one of the key parts of Forma Sonus, as this is what allows incoming sound data to have an influence on visual elements. Audio can enter the system through three different input methods, which are file upload, microphone input and line in input from an external audio device. Once the audio source is selected, it is passed into the useAudioAnalyzer hook, where the Web Audio API is used to handle the processing.

Inside this hook, an AudioContext is created and the incoming audio is connected to an AnalyserNode. The analyser is responsible for reading the sound signal and applying FFT which converts the audio from a time-based signal into frequency data (Lyons, 2010). This makes it possible to measure how much energy is transferring across different parts of the sound spectrum.

The frequency data is then divided into three main bands lows, mids and highs. These values are normalised to a range between 0 and 1 so they can be used more easily by the visual components. Along with the full frequency array, these band values are stored in the audioData state and updated continuously while the application is running. This processed audio data is then passed into the active visual layers, where it is used to influence different visual properties such as movement, scale and colour.

4.2.2 Data Flow

Data moves through Forma Sonus starting from the audio input and ending at the final visual output on screen. The system is mainly driven by two factors: incoming audio data and the user's interaction with the controls. An overview of this process is shown in Appendix D *(Figure D.1).*

At the start of the flow, audio enters the system through one of the three input methods. As shown in the diagram, all of these inputs feed into the same audio processing stage. This is handled by the useAudioAnalyzer hook, where the Web Audio API is used to process the incoming signal. Inside this

stage, an AnalyserNode performs FFT on the audio, which converts it into frequency data. The data is then split into lows, mids and highs and normalised before being output as audioData.

Once the audio has been processed, the audioData goes into App.jsx, w hich acts as the central state of the application and all the data will be branched out from here. This can be seen in the diagram where both audio data and layer configuration are managed together. From here, the audio data is passed down to each active visualisation layer as props (audioData.lows/mids/highs). Each layer uses this data differently, depending on its design, and updates its animation every frame. This is what allows the visuals to react in real time to the sound.

At the same time, user input flows into the system through the UI controls shown on the left of the diagram. When a user changes something like colour, speed or sensitivity, that change is sent back up to App.jsx, updating the layer configuration state. The updated settings are then passed back down to the visual layers, which adjust their behaviour straight away. This creates a continuous loop between user input, state updates and visual output.

The diagram also shows how Present Mode works as a separate part of the flow. When it is activated, the full layer state is sent from the main application to a new window using a BroadcastChannel. The present view receives this state and renders the same visuals, just without any of the interface controls only the canvas output.

Overall, the data flow is designed so that audio input, user interaction and rendering are all connected. This lets the system update continuously and respond immediately, which is important for keeping the visuals in sync with the audio.

4.2.3 Component Structure

The component structure of Forma Sonus is based on React, with App.jsx acting as the main component that manages the overall state of the application. This includes the active layers, their settings, the selected audio input and preset data. State is passed down to other components as props, keeping different parts of the system connected. The overall structure is shown in Appendix D *(Figure D.2)*.

As shown in the diagram, the application is divided into three main types of components: audio components (AudioUploader, MicrophoneInput, LineInInput), visual components (individual visual layers such as FrequencyBars, ParticleWave, etc.), and UI components (Controls Panel). Audio components handle the different input sources and work with the useAudioAnalyzer hook to process the sound data. Visual components are implemented as separate layers, each receiving audioData and settings as props. UI components allow user interaction, enabling settings to be adjusted through controls such as sliders and buttons.

When a user updates a setting, the change is passed back up to App.jsx, where the state is updated and then sent back down to the relevant visual layer. This structure keeps the application organised and makes it easier to manage how data flows between different parts of the system.

4.3 Data Persistence Design

For versioning within a session, like saving settings of the current layer setup, the app stores these versions in React state. These aren't saved to disk, so if the page is refreshed, the session versions are lost. This choice was made on purpose to keep things simple and prevent cluttering localStorage with temporary data, If a user wants to keep their current setup, they do have the option to save it.

The "Export Layer Session" option, which allows users to download their current setup as a JSON file. This is handy for sharing visuals or keeping a backup. The exported files store the configuration of each layer, including properties such as the selected visual type, parameter values (e.g. scale colour). It does not include any audio data or uploaded files, meaning users have to reselect an audio source when reloading a session. An example of the exported JSON structure can be seen in Appendix A (Figure A.8).

Overall, the method in which data is saved in Forma Sonus is simple. It just uses what is already built into the browser, so there is no need for a server or database. This makes everything quick and easy, and you do not need to log or sign in to save their work. If the project ever got bigger, maybe cloud saving or user profiles could be added.

4.4 Visual Rendering Architecture

Visual rendering in Forma Sonus is handled using React Three Fiber, where all visuals are drawn inside a Three.js canvas. Each visual is implemented as its own component, allowing multiple layers to be active at the same time.

Visuals are updated using the useFrame hook, which runs every frame. Inside this loop, audioData is read and used to update properties such as scale, position and colour. This allows the visuals to respond continuously to changes in the audio.

Different visual types use the audio data in different ways. For example, frequency bars use the frequency array to control height, while other visuals use values like lows, mids and highs to affect movement or scaling.

Overall, this setup allows visuals to update smoothly in real time while responding to both audio input and user settings.

4.4.1 React Three Fiber Integration

React Three Fiber is used to integrate Three.js into the React application. It enables 3D objects to be written as React components, which makes it easier to organise visuals alongside the rest of the interface. Each visual layer is rendered inside a canvas and updated based on state and audio data. This approach keeps the visual system consistent with the way the rest of the application is structured.

4.4.2 WebGL Rendering Pipeline

Rendering in Forma Sonus is done using WebGL with the help of Three.js. When a visual is created, things like its shape, colour, and any transformations (like scaling or rotation) get sent to the GPU, which handles all the heavy lifting and draws everything on the screen. This setup means you can have multiple visuals and layers running at the same time without things slowing down too much. Using WebGL is what makes it possible to keep everything running smoothly, even when the visuals get a bit more complex.

4.4.3 Animation Loop

Visual updates are handled through the animation loop using the useFrame hook. This runs every frame and is used to update visual properties such as scale, position and colour. The audioData values are read inside this loop, allowing visuals to respond continuously to changes in sound. Because this runs in sync with the render cycle, the visuals appear smooth and responsive.

4.5 Visual Systems

Forma Sonus comes with a range of built-in visuals, each reacting to audio in its own way. Each visual is set up as a separate component, making it easy to mix and match them using the layer system. Some settings are shared between visuals, such as colour or sensitivity, but many also have their own unique controls. For example, the Particle Wave visual lets you adjust parameters like the number of particles or how much they wobble, which isn't available in the other visuals. This way, each visual can be tweaked to suit its own style, rather than forcing everything to use the same set of controls.

4.5.1 Frequency Bars

The frequency bars visualisation was one of the first ones which were designed and is one of the most recognisable type of audio visualiser. The idea is straightforward: a row of bars, each representing a different part of the frequency spectrum, that grow taller when there is more energy in that frequency range. See Appendix C (*Figure C.1)* for an example of this visual.

The implementation creates a group of 3D box meshes arranged in a line. Each frame, the full frequency array from the audio analyser is sampled at evenly spaced intervals across the bars, so each bar corresponds to a different frequency range. The height of each bar is set directly from the frequency value at that point, multiplied by a sensitivity setting that the user can adjust.

Beyond the basic version, several extra options were added. The layout can be switched between linear (the default row of bars) and circular (bars arranged in a ring). Rounded tops and opacity control were also added as extra customization options.

4.5.2 Particle Wave

The Particle Wave visualisation was designed to be a more abstract and visually striking option compared to the frequency bars. Rather than showing data in a structured way, it uses a large number of small particles to create either a liquid sphere shape or an ambient dust field, both of which react to the audio in different ways. The positions of all particles are generated once on load using useMemo, which means the initial geometry is only computed once rather than on every render. For the liquid sphere mode, particles are distributed across the surface of a sphere using spherical coordinates. For the dust field mode, particles are scattered randomly within a box-shaped volume. See Appendix C (*Figure C.2* and *Figure C.3*) for an example of these visuals.

Each frame, the particle positions are updated based on a combination of time, per-particle random seeds, and the current audio data. The wobble amount controls how much the sphere surface distorts, and this can optionally be driven by a specific frequency band. For the dust field, the particles drift and shimmer at a rate influenced by the audio. A smoothing control allows the user to make the movement feel more fluid or more reactive, depending on their preference.

The particle system uses Three.js Points geometry, which is great for rendering large numbers of particles in WebGL without the expense of individual mesh objects, which would have a negative effect on performance. GLSL (OpenGL Shading Language) is used to control the appearance of each particle. Custom vertex and fragment shaders render every particle as a glowing, circular point with smooth edges, colour and transparency. While all particle movement and animation are handled in Javascript, the GLSL shaders are responsible for visually enhancing the particles as it was found that the original three.js particles were not visually up to standard. The GLSL was responsible for creating a soft glow and smooth fading between particles. The shader code was adapted from several different online resources and refined with the assistance of Copilot, rather than being entirely created from scratch. For an example of this GLSL shader code see Appendix A (Figure A.6).

4.5.3 3D Audio Shapes

The Audio Shapes visualisation displays a single 3D geometric shape in the centre of the canvas that pulses and rotates in response to the audio. The shape type can be selected by the user from a range of built-in Three.js geometries, including cube, sphere, torus, icosahedron, cylinder, cone, torus knot, octahedron, dodecahedron, and tetrahedron. Each frame, the shape's scale and rotation are updated based on the audio data. The user can independently choose which frequency band drives the rotation and which drives the scale, so for example the shape could pulse on the bass while spinning faster during high frequency content. For an example of this visual, see Appendix C (*Figure C.4).*

The scaling formula adds the audio band value multiplied by sensitivity on top of a base scale, so the shape always has a visible size even during silence. Wireframe mode is supported, as well as an emissive glow effect using Three.js material properties and an optional outline effect achieved by rendering a second slightly enlarged version of the shape behind the main one. Interestingly enough, a mistake during development led to an unexpected but visually striking feature: when wireframe mode was enabled and the shape was scaled up significantly, the resulting thin, rotating lines created an intricate pattern in the background. This accidental discovery added a unique and appealing visual element to the application, and was intentionally kept as a design feature.

4.5.4 Shape Grid

The Shape Grid is designed to act as a background layer for the other visualisations. Rather than being a standalone focal point, it fills the entire canvas with a repeating grid of flat 2D shapes that pulse and flash in response to the audio. The grid is calculated based on the canvas dimensions and the selected tile size and gap settings. Supported shape types include square, triangle, hexagon, circle, pentagon, and diamond. The geometry for each tile is generated once using useMemo and all tiles share the same geometry instance, which keeps the memory footprint low.

Because this layer is intended as a background, specific depth rendering settings were applied to prevent it from interfering with 3D layers in front of it. The material has depthWrite set to false and the render order is set to a lower value than other layers, which forces it to always draw behind everything else in the shared canvas.

The grid reacts to audio by pulsing its scale slightly on bass hits and adjusting its opacity based on the high frequency content. A colour reactive mode is also available which randomly flashes individual tiles to colours from a configurable palette when a peak is detected in the audio. For an Example of this visual see Appendix C (*Figure C.5).*

4.5.5 Waveform Visualisation

The Waveform visualisation displays the audio frequency spectrum as a continuous line that rises and falls across the screen. It was designed to be a clean and minimal option that gives a clear real time view of the audio signal shape. For an example of this visual, see Appendix C *(Figure C.6).*

The line is made up of 128 points spread evenly across the horizontal width of the canvas. Each frame, the frequency array from the analyser is sampled at 128 evenly spaced indices and the height of each point is set from the corresponding frequency value. A smoothing control was added which blends each point's current value with its value from the previous frame, which makes the movement feel more fluid rather than jumping sharply with every frame.

The waveform style can be changed between solid, dashed, and dotted using Three.js line distance rendering. An auto rotate option was also added which slowly rotates the whole waveform group on the Y axis, giving it a more dynamic look when displayed in present mode on a second screen.

4.5.6 Text Visualisation

The Text Visualisation renders a custom 3D text string in the centre of the canvas that pulses and optionally rotates in response to the audio. The design intention was to allow users to display a song title, artist name, or any other text as part of their visual setup. The text is rendered using the Text3D component from the drei library (A helper library built on top of react Three Fiber that provides additional components for working with 3D scenes), which takes a typeface JSON font file and extrudes the text into a 3D mesh. Several fonts are included in the application, including Helvetiker, Audiowide, Orbitron, and Press Start 2P, giving users a range of styles from clean, minimal to maybe more alternative funky options. For an example of this visual see Appendix C *(Figure C.7)*.

Each frame, the scale of the text is increased proportionally to the low frequency band value, so the text pulses outward on bass hits. An optional rotation mode continuously spins the text on the Y axis, with the speed increasing when there is more bass energy. The font, font size, letter spacing, depth of the 3D extrusion, and text content are all configurable through the controls panel.

4.6 Layer-Based Composition System

The layer-based composition system allows multiple visuals to be combined together at the same time. Each layer represents a single visual component, such as frequency bars, particle visuals or 3D shapes, and works independently from the others. For an example of this, see Appendix C *(Figure C.8).*

Each layer stores its own settings, including colour, scale, sensitivity, and behaviour. This means changes made to one layer do not affect any of the others, allowing different visual types to be adjusted separately. Layers can also be added or removed depending on what the user wants to create.

When multiple layers are active, they are rendered together within the same canvas. This allows different visual styles to be stacked and combined into a single output. For example, a waveform visual could be used alongside particle effects or 3D shapes to create a more complex result.

4.7 User Interface Design

The Forma Sonus user interface is designed to let users adjust visual settings while the application is running. Because everything updates in real time, the interface focuses on making changes happen instantly, allowing users to experiment with different settings without any delay. The main focus is on the visualisation, and on the left side of the screen is where the user interacts with the parameters and features. See Appendix E (Figure E.2) for an example of the UI.

The design and layout was inspired by existing visual tool interfaces such as modVisuals and particles.casberry with their main focus aswell being on the visual and the UI then surrounds it. The UI is built using Chakra UI, which provides ready made components like sliders, buttons and layout elements. This allowed for fast development of a consistent and visually nice application without spending a lot of time on styling.

The interface is also split into two modes, Design Mode and Layer Mode. Design Mode is more focused on working with one visual at a time, while Layer Mode is used when combining multiple visuals together. Splitting it like this made the layout feel less cluttered and easier to follow, especially when more is happening on screen.

4.7.1 Control Panel

Each visual in Forma Sonus includes its own set of controls, which are grouped into the panel on the left side of the interface. The panel allows users to adjust different parameters depending on the selected visual. Common controls include sliders for things like scale and audio sensitivity, colour pickers for appearance, and toggle switches for enabling or disabling certain behaviours like rotation or audio reactivity. For an example of the control panel see Appendix E (Figure E.1).

Alongside these parameters, the control panel also includes other options related to the overall setup. This includes selecting the audio input source, such as file upload, microphone or line in, as well as saving the current configuration as a preset. These features are placed within the same area so that users can manage both the visual behaviour and the overall setup without needing to switch between different parts of the interface.

Some visuals also include their own specific controls that are not shared across all types. For example, the 3D shape visual allows users to adjust scale across the X, Y and Z axes, either by using sliders or by entering values directly for more precise control. The text visual also includes an input field, allowing users to type and update the displayed text directly from the panel.

4.7.2 Interaction Design Decisions

The interaction design of Forma Sonus focuses on keeping everything as simple as possible and using familiar interactable elements. One of the main decisions was to avoid using complex menus where possible. Instead of dropdown menus, button grids are used for selecting options such as visual types or frequency bands. This makes it easier to see all available options at once without needing extra clicks. Another important decision was to keep all changes in real time. When a user adjusts a slider or toggles a setting, the result is shown immediately in the visuals. This removes the need for any confirm or apply actions, which helps keep the workflow faster and more suited to experimentation.

Overall, the interaction design was kept as direct as possible, so users can focus on changing visuals and seeing the results straight away, rather than navigating through different elements in the interface.

4.7.3 Accessibility Considerations

Accessibility was considered in the design of the interface, mainly by keeping the layout simple and easy to understand. Controls are clearly grouped and labelled, which helps users find and adjust settings quickly. The use of sliders, buttons and colour pickers is a common and familiar element that also makes interaction more direct, rather than relying on more complex inputs.

Because Forma Sonus runs in the browser, it can be accessed without installing any additional software, which makes it easier for users to try the application on different systems. However, the interface was primarily designed and tested for desktop use, so it may not perform as well on smaller screens or on touch devices.

5. Implementation

5.1 Development Environment

Forma Sonus was developed using Visual Studio Code as the main code editor. The application itself was built with JavaScript and React, with React Three Fiber used for handling the visual rendering. Three.js is used for working with 3D graphics, while the Web Audio API is used to detect and analyse audio input.

Vite was used as the development tool to run the project locally. Vite provides a quick development server with hot module reloading, making it easier to see and make changes quickly while the application is being built.

GitHub served as the version control for the project. It enabled tracking of changes over time, managing various code versions and switching to previous states when needed.

Vitest and React Testing Library were used for unit and component testing and Playwright was used for end-to-end browser testing. Google Chrome was the primary browser for both manual and automated testing due to its strong support for WebGL and the Web Audio API. NPM was also used to manage dependencies and to install libraries. Chakra UI was used to build the interface; it had prebuilt components that matched the style needed, which helped speed up development.

Overall, the development environment was kept relatively simple with tools that work well together for building and testing.

5.2 Application Architecture

The application is a single page React app built and served using Vite (Vite, 2023). The entry point is main.jsx, which handles one important routing decision before anything else renders. It checks the URL query string for a present parameter if it is there, it mounts the PresentView component, otherwise it mounts the main app component wrapped in Chakra UI's ChakraProvider. This is how the Present Mode window works it is the same built application, just loading a different root component depending on the URL.

The main App component acts as the central hub of the application. It holds all the application state and is responsible for connecting the audio processing, the UI controls, and the visualisation rendering together. Data flows downward from App to child components as props, and changes from the UI flow back up via callbacks. The specifics of how state is managed are covered in section 5.4.2.

5.3 Audio Analysis Implementation

The audio analysis side of the application is one of the more technically involved parts of the project. The goal was to take a live audio signal from whatever source the user provides and convert it in real time into a set of normalised values that the visualisations can use to react to the music. This is all handled through the browser's built in Web Audio API.

5.3.1 Web Audio API

Web Audio API works as a node graph, audio flows from a source node through one or more processing nodes and eventually out to a destination, which is usually the speakers. In this application, the first thing that gets created is an AudioContext, which is the main entry point for everything in the Web Audio API. Without it nothing else can be set up. From there, an AnalyserNode is created and connected into the graph, which is what does the actual frequency analysis. See (*Figure A.1*) for an overview of how the AudioContext and AnalyserNode are initialised for real-time audio analysis.

From the applications three supported methods of audio input (File upload, Microphone and Line in) end up feeding into the same internal connectSource function which attaches them to the analyser node in the same way. The only difference is that for microphone and line-in inputs, the audio is deliberately not connected through to the speakers. This prevents a feedback loop where the microphone picks up its own output and creates an increasingly loud echo.

5.3.2 FFT Processing

Once the audio source is connected to the AnalyserNode, the node continuously performs a Fast Fourier Transform on the incoming signal. FFT is a mathematical algorithm that converts a time-domain signal (raw audio samples over time) into a frequency domain representation (how much energy is present at each frequency at a given moment). This is what makes it possible to know how loud the bass is versus the treble at any point in time.

The fftSize is set to 2048, which means 2048 audio samples are analysed at once per frame. This produces 1024 frequency bins as output each bin representing a small bit of the frequency spectrum. At a standard sample rate of 44,100Hz, each bin covers approximately 21Hz of frequency range. To actually read the data out, the getByteFrequencyData method is called each frame, which fills a pre-allocated Uint8Array with the current frequency values. Each value in the array is an integer between 0 and 255, where 0 represents silence in that frequency range and 255 represents the maximum detectable amplitude. Importantly, this array is pre-allocated once when the audio context is initialised rather than creating a new array every frame, which avoids unnecessary memory allocation during the animation loop. See Appendix A (*Figure A.2*) for an overview of the FFT processing within the code.

This whole process runs inside a requestAnimationFrame loop, which ties the analysis to the browser's rendering rate at roughly 60 times per second. This is fast enough to keep the visuals feeling responsive and in sync with the audio.

5.3.3 Frequency Band Mapping

Having 1024 individual frequency bins is more than the visualisations need. Most of them just need to know the general energy level in the bass, mid, and high ranges rather than the exact value at every frequency. So, after the FFT data is read each frame, it gets summarised into three bands.

Rather than hardcoding which bin numbers correspond to these frequencies, the boundaries are calculated dynamically from the sample rate using the formula:

$$\text{binIndex} = \frac{\text{frequencyHz}}{\text{sampleRate}/2} \times \text{binCount}$$

For an overview of this formula in my code being implemented, see Appendix A (*Figure A.7).*

This means the band boundaries are always accurate regardless of what sample rate the user's audio hardware is running at. The calculation is also only done once per session and the result is cached, so it is not being recalculated on every frame. Once the boundaries are known, a single pass is made through the frequency array to accumulate the sum of values in each band. Each band's sum is then divided by the number of bins it contains and divided by 255 to produce a final normalised value between 0 and 1. These three values lows, mids, and highs are what get published as React state and passed down to the visualisation components, where they drive animations like scaling, rotation speed, and colour changes.

5.4 Custom React Hooks

React hooks are a way of extracting reusable logic out of components and into standalone functions. In this project, custom hooks were used to keep complex logic separate from the UI, which made the code easier to manage.

5.4.1 useAudioAnalyzer

useAudioAnalyzer is a custom hook that contains all of the Web Audio API logic for the application. As shown in Appendix A (*Figure A.2)*, the audio analysis loop inside the hook sets up and manages the AudioContext and AnalyserNode, reads FFT data every frame, and performs the frequency band calculations. The hook returns an audioData object containing the normalised lows, mids, and highs values along with the full frequency array, as well as functions for starting and stopping each audio input type. App.jsx calls this hook once and distributes the returned data to the rest of the application. This means the audio processing logic exists in exactly one place, and everything else just consumes the output.

5.4.2 State management strategy

All application state is stored in a single component, App.jsx. This includes the list of active layers and their settings, the current preset data, saved version history, audio input state, background colour, and other UI states like which modals are open. The pattern used throughout the app is props down, callbacks up. State lives in App.jsx, gets passed down to child components as props, and when a child needs to update something, it calls a callback function that was also passed down as a prop. For an

example of top level state management in App.jsx using Reacts useState for layers,presets and version history see Appendix A (*Figure A.3*).

5.5 Visual Component Implementation

Each visualisation in the application is its own self-contained React component. They all follow a similar structure, they receive audioData and their own settings as props, and they use that data to update their animations every frame. All of the visualisations render inside a single shared <Canvas> element provided by React Three Fiber, which sets up a WebGL rendering context. Within that canvas, each active layer mounts its corresponding visualisation component, and they all share the same 3D scene and camera.

The animation logic in each component runs inside useFrame, which is a hook provided by react three fiber that fires a callback on every render frame. Importantly, useFrame runs outside of React's normal rendering cycle, which means updating a Three.js object's position or scale inside it does not trigger a React re-render. See Appendix A (*Figure A.5*) for an example of a visual component that receives audio data and settings as props, and updates its animation frame using the useFrame hook.

5.6 Layer Management Logic

The layer system is built around a simple array stored in App.jsx state. Each entry in the array is an object that contains a unique ID, a type string identifying which visualisation it is, and a settings object holding all of that layer's configurable values. When the canvas renders, it maps over this array and mounts the appropriate visualisation component for each layer, passing in its settings.

Adding a new layer appends a new object to the array with a generated ID and a set of default settings for that type. Removing a layer filters that entry out of the array by its ID. Updating a layer's settings finds the matching entry by ID and replaces its settings object with the updated values, leaving all other layers untouched. For an example of how layer management logic in App.jsx showing how layers are added, removed, and updated in state see Appendix A (*Figure A.4*).

5.7 Control Component Implementation

Each visualisation type has a paired control component that provides the UI for adjusting its settings. These are all kept in a separate controls folder, with names like FrequencyBarsControls and AudioShapesControls, and they are rendered in the settings panel when the user selects a layer.

Each control component receives the current settings for its layer as a props object and an onChange callback. When the user moves a slider or flips a switch, the component calls onChange with just the property that changed, and App.jsx handles merging that update into the full layer settings object.

5.8 Use of AI Tools During Development

AI tools were a huge help during this project, especially since much of the tech was new at the start. Github co pilot and ChatGPT was used frequently to understand how different libraries worked, to get explanations for errors, and to brainstorm ideas when stuck. These tools were useful when working from existing Three.js examples found online as inspiration. They helped break down how different visuals worked, making them easier to understand and adapt to the requirements of Forma Sonus. This made it possible to build similar visuals, even when the original examples were quite different.

AI tools were also helpful when mapping audio data to different visual parameters. When it wasn't clear how a specific effect should respond to the music, they were used to get suggestions or example code, which could then be adjusted to better fit the intended behaviour. GitHub Copilot was particularly useful when working with GLSL shader code for the particle-based visuals. It provided suggestions and helped fill in parts of the code when needed, which made working with shaders more manageable.

Both ChatGPT and Copilot were also used when writing the Playwright testing code. They helped by providing examples and suggestions for structuring tests and interacting with the interface, making it easier to set up automated testing. AI tools were not only used for coding, they were also used to assist with creating diagrams for the report, such as the application architecture diagram, which helped present how different parts of the system connect together.

Overall, AI tools made the development process much smoother. They were especially useful for learning on the go, building and customising visuals, getting testing set up, and getting unstuck when things got confusing. It was like having a second pair of eyes to check things or explain concepts when needed.

5.9 Challenges Encountered

Initial Project Ideas

One of the main challenges during the project was adjusting the original idea into something that was actually achievable within the time available. As mentioned earlier, the project first started as a TouchDesigner-based idea, but it quickly became clear that it would be too resource intensive and too large in scope to complete. Another idea was explored after that, but it ran into similar issues and was also dropped. In the end, this led to the current version of the project, which was shaped by those earlier limitations and a focus on creating something more accessible for users.

Time

Another challenge was completing the project within the remaining time after changing direction. Much of the time had already been invested in earlier ideas, leaving less time for the final version. Ultimately, careful planning and prioritising the main features became crucial to ensure the core parts of the application were finished correctly.

Performance

One of the earliest performance issues encountered was caused by how frequently React was re-rendering when audio was active. Because audioData is React state that updates 60 times per second, any component connected to it would also re-render at that rate. When the controls panel was open, this caused noticeable frame drops because the entire settings UI was being re-evaluated on every audio frame even though nothing in the controls had changed. The solution was to apply React.memo to components that do not need to update with the audio, which tells React to skip re-rendering them unless their own props change. This helped performance marginally; however, the issue still showed at times.

Managing State Across Layers

Managing multiple layers, each with its own settings, made state management a bit tricky. Whenever a setting needed to be changed for one layer, it was important to ensure it didn't affect any of the others. This meant carefully updating the state array every time something changed, the code had to find the right layer by its ID and update only that one, leaving the rest alone. It took a bit of trial and error to get this working smoothly, but it was worth it to keep everything independent.

BroadcastChannel Synchronisation

Keeping the Present Mode window in sync with the main application was tricky to get right. The BroadcastChannel API itself is straightforward, but the challenge was ensuring that every state change in the main window was broadcast correctly and that the present view always had a complete, up-to-date snapshot of the layer state rather than partial updates. The solution was to broadcast the entire relevant state object on every change, so the present view always receives a full picture rather than trying to merge incremental updates.

Fullscreen Compatibility

The original implementation used native browser dialogs (window.confirm and window.prompt) for tasks such as naming presets and confirming version restores. These dialogs fail when the browser is in fullscreen mode (the browser exits full screen before showing them), which is a poor experience. These were replaced with custom Chakra UI modals that render within the application and work correctly regardless of whether the browser is in fullscreen mode.

6. Testing and Evaluation

6.1 Testing Strategy

A lot of the core functionality, including audio analysis, WebGL rendering, and real-time reactivity, is difficult to test automatically because it depends on live audio input and visual output. Because of this, the testing approach was split into several parts rather than relying on a single automated method.

Unit testing was used where possible to check smaller pieces of logic, while end-to-end testing was used to replicate user interaction through the interface. Manual functional testing was also carried out to make sure key features worked, such as audio input, layer control, and preset handling. In addition to this, usability testing was used to gather feedback on how easy the application was to use and understand, particularly from users who were not familiar with the system.

6.2 End-to-End Testing (Playwright)

End-to-end tests were written using Playwright (Microsoft, 2023), which is a browser automation framework that controls a real browser and interacts with the application the same way a user would. This was chosen over unit testing as most of the things to test in the application are behaviours the user would encounter, such as the app loading, whether they can switch modes, and whether they can save a preset. All 10 tests passed, see *Appendix B* (*Figure B.1)*

One issue that came up during testing was that the preset save flow originally used window.prompt, which Playwright can intercept with a dialogue handler. When that was replaced with a Chakra UI

modal to fix the full-screen compatibility issue described in section 5.9, the tests had to be updated to interact with the modal instead: find the text input, fill it in, and click the confirm button. This was a good example of how a change in one part of the application had a knock-on effect on the tests.

6.3 Functional Testing

Functional testing was carried out manually to verify that each feature of the application worked correctly. These tests covered areas which automated tests could not like audio reactivity, visual behaviour and the present mode window.

Each feature was tested by interacting with it directly in the browser and checking that the output matched what was expected. See Appendix B *(Figure B.2)* for results

6.4 Unit Testing (Vitest)

Unit tests were written using Vitest and React Testing Library to test UI components in isolation separate from the full application, whereas the end to end tests verify the application works as a whole in a real browser, unit tests focus on smaller pieces of functionality. Five component tests files were written covering three visual control files, which looked into features such as the 3D shapes settings panel, rendering all shape type buttons or displaying the right sensitivity value.

The two additional test files covered the AudioUploader and MicrophoneInput modules. These tests verified that the upload button is disabled when a file is already uploaded. In MicrophoneInput, they checked that things such as the button label toggles between “Use Microphone” and “Stop Microphone” depending on the current state.

In total, 30 unit tests were run across 5 test files, all of which passed. See Appendix B *(Figure B.3)* for a screenshot of results.

6.5 Usability Testing

Usability testing was carried out by having a small number of people who were supplied a link to the application (which was hosted on vercel) and specific instructions for them to carry out. After using the app, they were asked to complete a Microsoft Form survey that asked about their experience, including how easy the interface was to use, whether any features were confusing, what they liked or disliked and would there be any improvements or additional features they would add. For an example of some of the user feedback from the survey see Appendix B *(Figure B.4)*

This method gave detailed feedback on both the user interface and the overall user experience. The survey responses revealed that most users found the controls straightforward and the app generally easy to navigate. Additionally, helpful suggestions were incorporated; for example, one user noted the inability to resize 3D shapes beyond a certain point, leading to the addition of a manual scale input for greater flexibility. Another user found that creating multiple saved visuals with identical names was confusing, so a validation check was implemented to prevent duplicates. Overall, this feedback contributed to improving the UI and making the tool more user-friendly for future users.

6.6 Performance and Debugging

Performance was monitored throughout development using an FPS counter built directly into the application. This displays the current frames per second in the corner of the UI, which made it easy to see in real time whether a change had improved or worsened rendering performance.

Debugging WebGL issues was more difficult than debugging regular JavaScript because errors in the rendering pipeline do not always produce clear error messages. This is where the use of AI came in and helped debug any issues I had by giving a detailed explanation of what was wrong. The depth buffer issue with the Shape Grid layer, where it was visually covering 3D layers despite being positioned behind them, was traced by systematically disabling layers and re enabling them one at a time until the conflict was brought down to the depth writing behaviour of the flat 2D geometry, it was never fully fixed though it had definitely improved from what it was before.

6.7 Evaluation Against Objectives

The idea was to let users create and customise real time visuals in the browser, and to have those visuals either animate on their own or react to audio input based on the user's preference. This approach worked quite well; you can experiment with all the visuals even without any audio. However, once you upload a track or activate your microphone, everything begins to respond in sync with the sound.

A key requirement was supporting different audio input types. Forma Sonus allows users to upload a file, use a microphone, or select a line-in device, making it suitable for both home testing and live setups.

Another important goal was to ensure that the visuals would update smoothly and feel responsive, without any noticeable lag. This was mostly achieved — the visuals animate in real time, and parameters like rotation speed or wave motion can be adjusted even when there is no audio. When audio is present, the visuals generally stay in sync with the beat, which was a major focus. However, as more layers are added, performance can start to dip, and the frame rate may drop, so the animation isn't always as smooth as intended. This is an area that could definitely be improved in future versions of the project.

Overall, Forma Sonus meets the main objectives that were set. There are areas for future improvement, such as more advanced mapping of audio to parameters or enhanced session saving, but the core features are present and functioning as intended.

7. Project Management

7.1 Development Methodology

The development of Forma Sonus followed an agile approach, but it was adapted to the reality of working under time constraints with regular supervisor meetings. At the start, a plan was made listing out all the main tasks and features that needed to be built. Instead of sticking strictly to that plan, the workflow was kept flexible so that priorities could change as the project progressed.

7.2 Sprint Planning

Sprint planning was carried out in collaboration with the project supervisor. Each week, meetings were held to review what had been completed so far and to decide together which tasks should be prioritised for the following week. Sometimes this meant sticking to the original plan, but often priorities would shift, for example, if a particular feature became more important or if something took longer than expected. After each meeting, there was a clear set of tasks to focus on for the week, and the next meeting would begin with a review of what had been finished and what still needed work. This approach made the process feel a lot like weekly sprints, even if it wasn't a formal Scrum setup.

7.3 Version Control Strategy

Git and GitHub were used throughout the project for version control. All changes were committed directly to the main branch, with regular commits made after finishing a feature or fixing a bug. Commit messages were kept as clear as possible to make it easy to look back and see what had changed over time. Using GitHub meant that the project was always backed up online and could easily be shared with the supervisor when needed. This approach kept the workflow simple and ensured that progress was always saved and accessible.

7.4 Tools Used

A few different tools helped keep the project organised:

- **GitHub:** For version control, code hosting, and tracking progress.
- **Trello:** At the start, a Trello board was set up with all the main tasks and features. This was useful for visualising what needed to be done, but as the project went on, the plan shifted away from the original board as priorities changed during supervisor meetings. Still, it was handy for keeping track of the bigger picture.
- **VS Code:** The main code editor, with extensions for React, JavaScript, and Git.
- **Vercel:** Used to host the live version of the app, which made it easy to share progress and get feedback.
- **Microsoft Forms:** For collecting usability feedback from testers.
- **ChatGPT and Copilot:** Used for help with tricky code problems, understanding new libraries, and brainstorming solutions.

7.5 Reflection on Development Process

Overall, the project management process was a mix of planning and flexibility. Having a big list of tasks at the start was helpful, but real progress came from the weekly meetings with the supervisor. Being able to discuss what was working, what wasn't, and what should be prioritised next made the whole process feel more manageable and less overwhelming. The "sprint like" weekly goals gave a sense of structure, but there was always room to adapt if something unexpected came up.

8. Conclusion and Future Work

8.1 Summary of Achievements

One of the main achievements of this project was the ability to adapt quickly in response to feedback and make a huge change in direction for the project. As mentioned before, the original concept focused on a TouchDesigner-based project, but discussions with the project supervisor and more research showed that this approach would be difficult to complete within the timeframe. Recognising this at the time was crucial, and deciding to pivot proved to be the best decision made during the process of the development of this final year project.

At the start, there was a steep learning curve, but the project still came together and Forma Sonus ended up matching the original vision relatively closely. The app lets users create and customise real time visuals, supports multiple audio input types, and has a flexible layer system for building up complex scenes. Seeing the project go from a rough idea to a working tool that people could actually use and test was a really proud moment.

8.2 Reflection on Learning Outcomes

This project was a huge learning experience, both technically and personally. On the technical side, it meant picking up new libraries and frameworks on the go. Technologies like Three.js, Web Audio API and figuring out how to get them all working together. There was a lot of trial and error, and plenty of times when things just didn't work the way they were supposed to. But working through those problems, asking for help or advice when needed, and not giving up made the end result feel even more rewarding. On the personal side, it was a lesson in time management, and staying motivated even when things got tough.

8.3 Limitations

While the project achieved most of its main goals, there were definitely some limitations. The biggest one was time. With more time, the app could have been polished further and had more features added, such as camera input. Performance also became an issue when stacking lots of layers, and some of the more advanced visualisations or features (like more complex 3D shapes or particle types) didn't make it in. There were also things like mobile optimisation and user accounts that would have been nice to include but just were not realistic in the timeframe.

8.4 Future Improvements

If there was more time to keep working on Forma Sonus, there are many ideas for future improvements:

- **More frequency to parameter mapping**: Have any audio band to control any setting, for more creative flexibility.
- **Camera input and effects:** Adding support for webcam input and visual effects based on video.
- **MIDI input:** Letting users control visuals with MIDI controllers for live performance.
- **Sign up and log in:** Adding user accounts so presets and sessions could be saved to the cloud.
- **Mobile optimisation**: Making the app work better on phones and tablets.
- **More base visuals:** Adding more built-in visual modules for users to customise and combine.

8.5 Final Conclusion

In the end, Forma Sonus proved to be a project worth being proud of. Especially considering the time crunch and the amount of new material that had to be learned along the way. It is a solid foundation for a creative tool, and with more time and development, it could become something even more powerful. The experience of building it was challenging but also very rewarding, and it showed that with enough determination, a big idea can be brought to life even when the odds seem stacked against you.

References

Bain, M. N. (2008). *Real-time music visualization: A study in the visual extension of music*. https://etd.ohiolink.edu/acprod/odb_etd/ws/send_file/send?accession=osu1213207395&disposition=inline

Cabello, R. (2010). *Three.js documentation*. https://threejs.org/docs/

Chakra UI. (2023). *Chakra UI documentation*. https://chakra-ui.com

Derivative. (2023). *TouchDesigner documentation*. https://derivative.ca

Foote, J. (2000). *Visualizing music and audio using self-similarity*. Proceedings of ACM Multimedia. https://www.musanim.com/wavalign/foote.pdf

Fujishiro, I., Nakayama, M., & Takakura, Y. (2018). *Visual analysis for the compositional process of composers in spectral school*. https://www.jstage.jst.go.jp/article/tievciieej/6/1/6_22/_pdf/-char/ja

Git. (2023). *Git documentation*. https://git-scm.com/docs

GitHub. (2023). *GitHub documentation*. https://docs.github.com

Graf, M., Opara, H. C., & Barthet, M. (2021). *An audio-driven system for real-time music visualisation*. https://arxiv.org/pdf/2106.10134

Khronos Group. (2011). *WebGL specification*. https://www.khronos.org/webgl/

Lyons, R. G. (2010). *Understanding digital signal processing* (3rd ed.). Pearson Education.

McCarthy, L., Reas, C., & Fry, B. (2015). *Getting started with p5.js: Making interactive graphics in JavaScript and Processing*. O'Reilly Media. https://openlab.citytech.cuny.edu/mtec1101-hd88-sp2022/files/2019/03/Make_Getting-Started-with-p5dotjs.pdf

Meta. (2023). *React documentation*. https://react.dev

Microsoft. (2023). *Playwright documentation*. https://playwright.dev

Mozilla Developer Network. (2023). *Web Audio API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

Nullsoft. (1997). *Winamp media player*. https://www.winamp.com/

Poimandres. (2019). *React Three Fiber documentation*. https://r3f.docs.pmnd.rs/getting-started/introduction

Vercel. (2023). *Vercel documentation*. https://vercel.com/docs

Vite. (2023). *Vite documentation*. https://vitejs.dev

Appendices

Appendix A – *Code Snippets*

```
// Create the main audio context and analyser node
const audioContext = new (window.AudioContext || window.webkitAudioContext)();
const analyser = audioContext.createAnalyser();
analyser.fftSize = 2048;

// Function to connect any audio source to the analyser
function connectSource(sourceNode, connectToOutput = true) {
  sourceNode.connect(analyser);
  if (connectToOutput) {
    analyser.connect(audioContext.destination);
  }
}
```

Figure A.1 – Initialising audioContext and analyserNode for real-time audio analysis

```
// Inside startAnalyzing in useAudioAnalyzer.js

// Get the current frequency data from the analyser node.
// This fills the dataArrayRef with the latest FFT values (0-255 for each frequency bin).
analyzerRef.current.getByteFrequencyData(dataArrayRef.current);

const data = dataArrayRef.current;      // The array holding the frequency data for each bin.
const binCount = data.length;           // Number of frequency bins (e.g., 1024 for fftSize 2048).

// Compute band boundaries (only once per session for efficiency).
if (!bandCacheRef.current) {
  // Get the sample rate (usually 44100 Hz).
  const sampleRate = audioContextRef.current?.sampleRate ?? 44100;
  // Calculate the frequency width of each bin.
  const hzPerBin = (sampleRate / 2) / binCount;
  // Determine the bin index where lows end (e.g., 250 Hz).
  const lowEnd = Math.max(1, Math.round(250 / hzPerBin));
  // Determine the bin index where mids end (e.g., 4000 Hz).
  const midEnd = Math.max(lowEnd + 1, Math.round(4000 / hzPerBin));
  // Cache these boundaries for future frames.
  bandCacheRef.current = { lowEnd, midEnd };
  // Create a Float32Array to store normalized frequency values (0-1).
  normalizedRef.current = new Float32Array(binCount);
}

const { lowEnd, midEnd } = bandCacheRef.current;
const normalized = normalizedRef.current;

// Initialize sums for each band.
let lowSum = 0, midSum = 0, highSum = 0;

// Loop through all frequency bins.
for (let i = 0; i < binCount; i++) {
  const v = data[i];            // Raw FFT value (0-255).
  normalized[i] = v / 255;      // Normalize to 0-1 for visualization.
  // Accumulate sums for each band.
  if (i < lowEnd) lowSum += v;
  else if (i < midEnd) midSum += v;
  else highSum += v;
}

// Calculate average (normalized) value for each band.
const lows = lowSum / lowEnd / 255;
const mids = midSum / (midEnd - lowEnd) / 255;
const highs = highSum / (binCount - midEnd) / 255;

// Update the state with the new audio data.
// - lows, mids, highs: average normalized values for each band
// - frequencies: full normalized frequency array for detailed visualizations
setAudioData({ lows, mids, highs, frequencies: normalized });
```

Figure A.2 – Audio analysis loop

```
// --- Layer Selection State (Layer Mode) ---
const [selectedLayerId, setSelectedLayerId] = useState(null);

// --- Layer Session Versioning ---
// Store named versions (snapshots) of the layers array
const [layerVersions, setLayerVersions] = useState([]);

// activePresetId: tracks which saved preset is currently loaded into the design mode controls.
// Used to show the "Editing Preset" banner and to push updated settings back to linked layers.
const [activePresetId, setActivePresetId] = useState(null);
```

Figure A.3 – State management logic

```jsx
// Store all layers in App.jsx state
const [layers, setLayers] = useState([]);

// Add a new layer (e.g., from a preset)
setLayers(prev => [...prev, {
  id: Date.now(),           // Unique ID for the layer
  type: preset.type,        // Visualization type
  settings: layerSettings,  // All configurable values for this layer
  // ...other properties
}]);

// Remove a layer by ID
function removeLayer(id) {
  setLayers(prev => prev.filter(l => l.id !== id));
}

// Update a layer's settings by ID
function updateLayer(id, updates) {
  setLayers(prev => prev.map(l => l.id === id ? { ...l, ...updates } : l));
}
```

Figure A.4 – Layer management logic

```jsx
import { useFrame } from '@react-three/fiber'

export const FrequencyBars = ({
  audioData,
  sensitivity = 1.0,
  colour = '#548c8c',
  // ...other settings as props
}) => {
  const groupRef = useRef()
  // ...refs for props

  useFrame(() => {
    if (!groupRef.current) return
    // Animation logic using audioData and settings
    const frequencies = audioData?.frequencies ?? null
    // ...update bar visuals based on frequencies and props
  })

  return (
    <group ref={groupRef}>
      {/* ...bar meshes */}
    </group>
  )
}
```

Figure A.5 – Visual component animation logic

```
<shaderMaterial
  ref={materialRef}
  transparent
  depthWrite={false}
  blending={THREE.AdditiveBlending}
  uniforms={uniforms}
  vertexShader={`
    // Per-particle size attribute
    attribute float size;
    // Pass alpha value to fragment shader
    varying float vAlpha;
    void main() {
      // Transform the particle position from model to view space
      vec4 mvPosition = modelViewMatrix * vec4(position, 1.0);
      // Project the position to screen space
      gl_Position = projectionMatrix * mvPosition;
      // Set the size of the point sprite (scales with perspective)
      // 420.0 is a scaling factor to make particles visible at typical camera distances
      // -mvPosition.z is the depth; max(1.0, -mvPosition.z) prevents division by zero/negative
      gl_PointSize = size * 420.0 / max(1.0, -mvPosition.z);
      // Set alpha to 1.0 (fully opaque, can be modulated in fragment shader)
      vAlpha = 1.0;
    }
  `}
  fragmentShader={`
    // Uniform color for all particles
    uniform vec3 uColor;
    // Uniform opacity for all particles
    uniform float uOpacity;
    // Alpha value passed from vertex shader
    varying float vAlpha;
    void main() {
      // gl_PointCoord is the coordinate within the point sprite (0,0 in bottom left, 1,1 in top right)
      // Center the coordinate system to (0,0) in the middle
      vec2 uv = gl_PointCoord - 0.5;
      // Compute distance from center (for circular mask)
      float d = length(uv);
      // Use smoothstep to create a soft edge for the particle (anti-aliased circle)
      float alpha = smoothstep(0.5, 0.0, d) * vAlpha;
      // Set the fragment color: RGB from uColor, alpha modulated by uniform opacity and calculated alpha
      gl_FragColor = vec4(uColor, alpha * uOpacity);
      // Discard fragments with very low alpha for performance and clean edges
      if (gl_FragColor.a < 0.01) discard;
    }
  `}
/>
</points>
)
}
```

Figure A.6 – GLSL shader code logic

```
const sampleRate = audioContextRef.current?.sampleRate ?? 44100
const hzPerBin = (sampleRate / 2) / binCount
const lowEnd = Math.max(1, Math.round(250 / hzPerBin))
const midEnd = Math.max(lowEnd + 1, Math.round(4000 / hzPerBin))
```

Figure A.7 – Frequency to bin formula

```
[
  {
    "id": 1777649201439,
    "presetId": 1777649194980,
    "label": "2",
    "type": "audioShapes",
    "settings": {
      "colour": "#ffffff",
      "sensitivity": 1,
      "audioReactive": false,
      "shapeType": "icosahedron",
      "size": 2.25,
      "scaleX": 1,
      "scaleY": 1,
      "scaleZ": 1,
      "wireframe": true,
      "emissiveColour": "#000000",
      "emissiveIntensity": 0,
      "shadow": false,
      "outlineEnabled": false,
      "outlineColour": "#ffffff",
      "outlineThickness": 0.05,
      "scaleBand": "lows",
      "rotationBand": "lows"
    },
    "opacity": 1,
    "enabled": true
  },
  {
    "id": 1777649198607,
    "presetId": 1777649042808,
    "label": "1",
    "type": "audioShapes",
```

JSON file length : 1,260 lines : 56

Figure A.8 – Exported JSON data

Appendix B – *Test Results*

```
PS C:\Users\JACKI\OneDrive\MajorProjectWebTool> npx playwright test --reporter=list

Running 10 tests using 1 worker

  ✓  1 [chromium] › e2e\app.spec.js:4:3 › App loads › shows the main UI on load (4.4s)
  ✓  2 [chromium] › e2e\app.spec.js:10:3 › App loads › shows audio upload and microphone buttons (2.1s)
  ✓  3 …e2e\app.spec.js:18:3 › Design / Layer mode switching › Design and Layers mode buttons are present (2.2s)
  ✓  4 [chromium] › e2e\app.spec.js:24:3 › Design / Layer mode switching › can switch to Layers mode (2.7s)
  ✓  5 [chromium] › e2e\app.spec.js:31:3 › Design / Layer mode switching › can switch back to Design mode (2.6s)
  ✓  6 … › e2e\app.spec.js:41:3 › Visual type selection in Design Mode › can select 3D Shapes visual type (2.9s)
  ✓  7 …› e2e\app.spec.js:48:3 › Visual type selection in Design Mode › can select Shape Grid visual type (3.6s)
  ✓  8 …] › e2e\app.spec.js:54:3 › Visual type selection in Design Mode › can select Waveform visual type (3.4s)
  ✓  9 …› e2e\app.spec.js:62:3 › Preset saving and loading › Save Preset button is present in Design Mode (2.2s)
  ✓  10 …um] › e2e\app.spec.js:67:3 › Preset saving and loading › can save a preset and see it in the list (3.4s)

  10 passed (41.5s)
```

Figure B.1 - Playwright end-to-end test results

Feature	Test Performed	Result
Audio file upload	Uploaded an MP3 and checked visuals reacted	Pass
Microphone input	Activated mic and spoke/played music near it	Pass
Visual reactivity	Checked lows/mids/highs drove animations in each visual type	Pass
Layer system	Added, reordered, and removed multiple layers	Pass
Preset save and load	Saved a preset and reloaded it in a new session	Pass
Duplicate preset name check	Attempted to save two presets with the same name	Pass — blocked
Present Mode	Opened present window and confirmed it synced with main window	Pass
Session export/import	Exported a session and re-imported it to restore state	Pass

Figure B.2 - Functional testing results

```
 ✓ src/test/AudioShapesControls.test.jsx (7) 4522ms
 ✓ src/test/AudioUploader.test.jsx (4) 1201ms
 ✓ src/test/MicrophoneInput.test.jsx (6) 2068ms
 ✓ src/test/ShapeGridControls.test.jsx (7) 4050ms
 ✓ src/test/WaveformControls.test.jsx (6) 2591ms

 Test Files  5 passed (5)
      Tests  30 passed (30)
   Start at  22:07:16
   Duration  16.45s (transform 1.60s, setup 1.84s, collect 32.67s, tests 14.43s, environment 11.
s, prepare 5.74s)
```

Figure B.3 - Unit testing results

2. Did you encounter any bugs or confusing moments?

- (●) Yes
- ( ) No

3. If yes please describe

- Couldn't make audio reactive movement slow enough for the song that I wanted. Couldn't scale up icosahedron as large as I would like. Consider having a number that I can enter manually for scale that has essentially no limit. State of saved layer visual doesn't update in layers view. There were some settings that I would have expected to have available on every visualisation e.g. scale, or rotation but on some they weren't available. It would be nice if I could map audio reactivity to any setting e.g. scale or rotation. I found that the workflow to create a multi-layered visual was clunky. If I made a change to a layer in design mode, it wouldn't update the changes in Layer mode. I also found that I could create multiple saved visuals with the same name which I think is confusing as they are global. I found that the "Save Layer Session" button exported the session as a JSON which I was not expecting and "Save Version" did what I was expecting "Save Layer Session" to do. Perhaps "Export Layer Session" is a better name for that? Also it doesn't seem to save the uploaded audio along with it, which I would find useful for loading a session for a VJ set.

6. Suggestions for saving and loading presets to make it clearer.

- Editing presets is quite confusing. If I load a preset and make changes to it, it appears that those changes aren't saved. But if I save it as a new preset of the same name, it creates a new preset rather than overwriting the existing one. I think it would be nice to know what current preset you're editing, and have it auto-save and update the layer view.

8. Any suggestions for improvement or additional features?

- What I like the most about this tool is speed to create a visual. It's much easier to create something from nothing than using a tool like touch designer. However, I think that there are quite a few bugs and UX issues that could be cleaned up before I trusted using it for a session myself. I also think that I would like to have a little bit more control. For example, I wanted to make certain aspects of a visual audio-reactive but couldn't, or only reactive to the low-end component of the song. I also wanted to scale up a visual very large but couldn't enter a custom scale figure. All of these are just small things but they would make the ability to create custom visuals that I have in my head much easier to do.

Figure B.4 – Survey Results

Appendix C – Visuals

Figure C.1 – Frequency Bars (Linear)

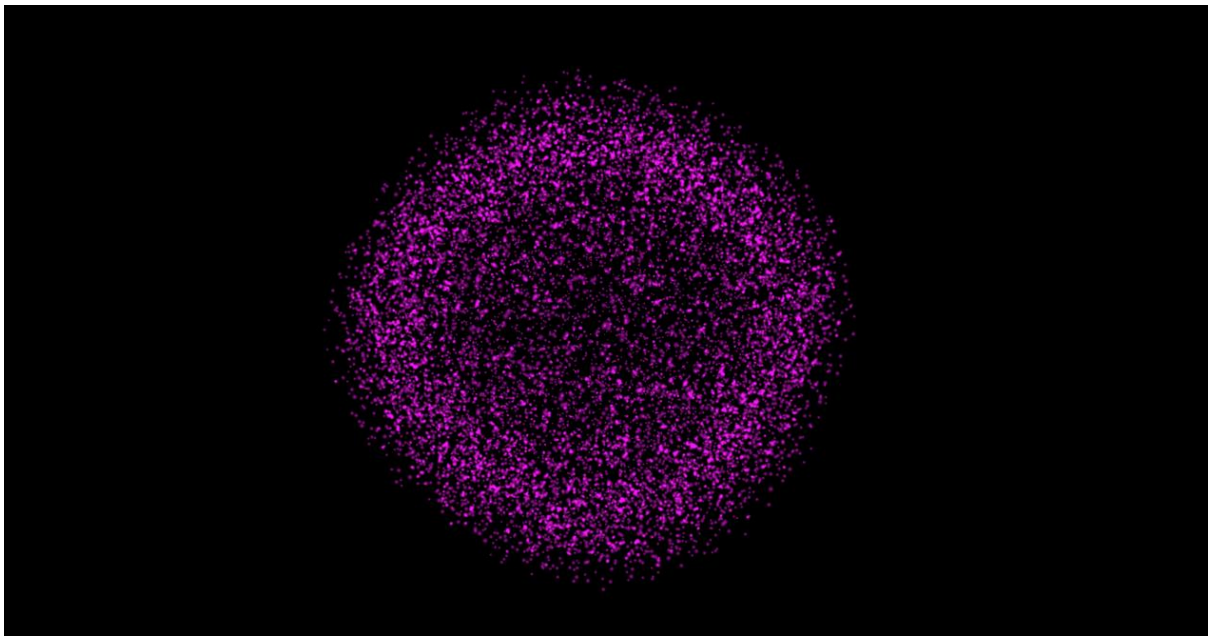

Figure C.2 – Particle Wave (Liquid sphere)

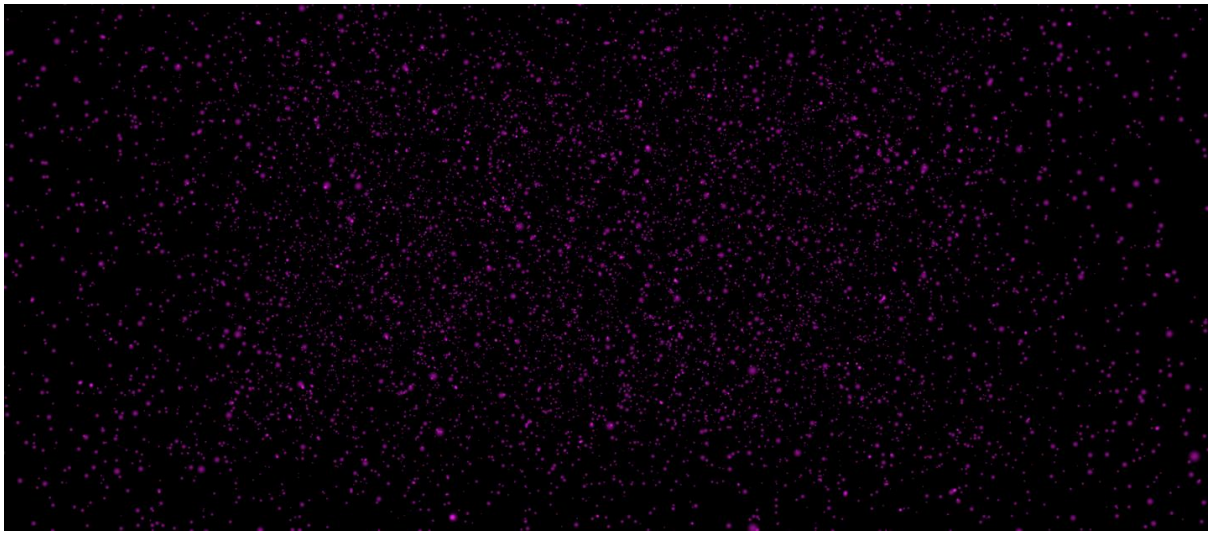

Figure C.3 – Particle Wave (Dust Field)

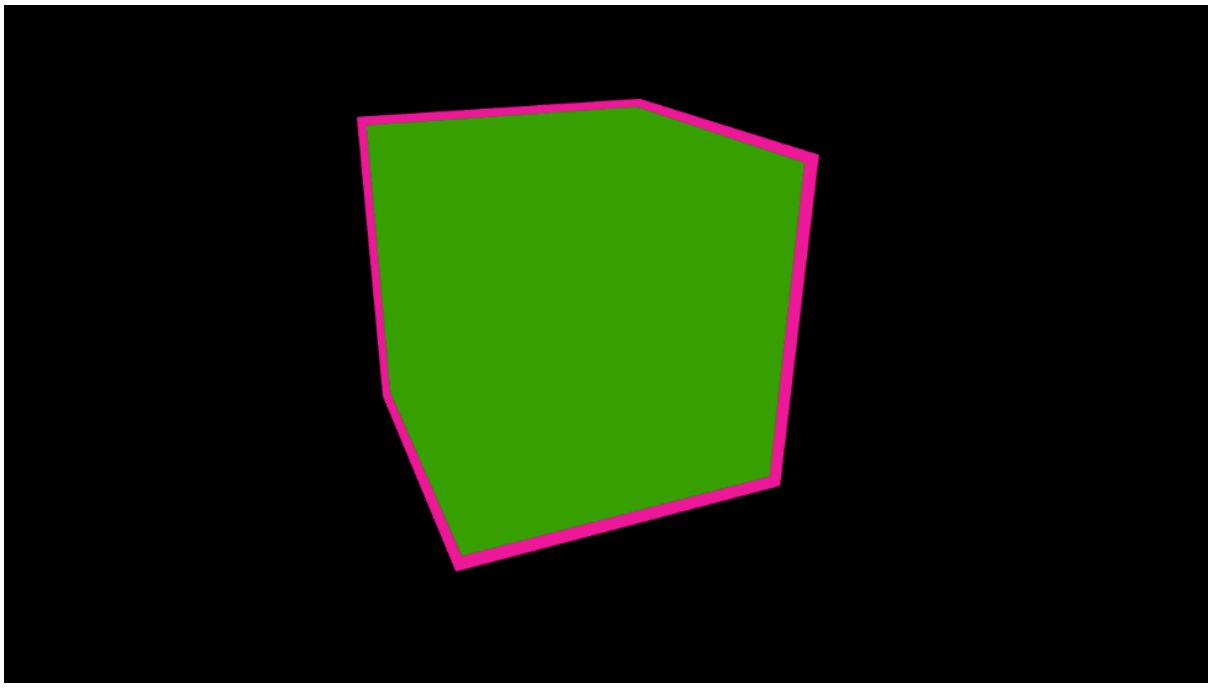

Figure C.4 – 3D Shapes (Cube)

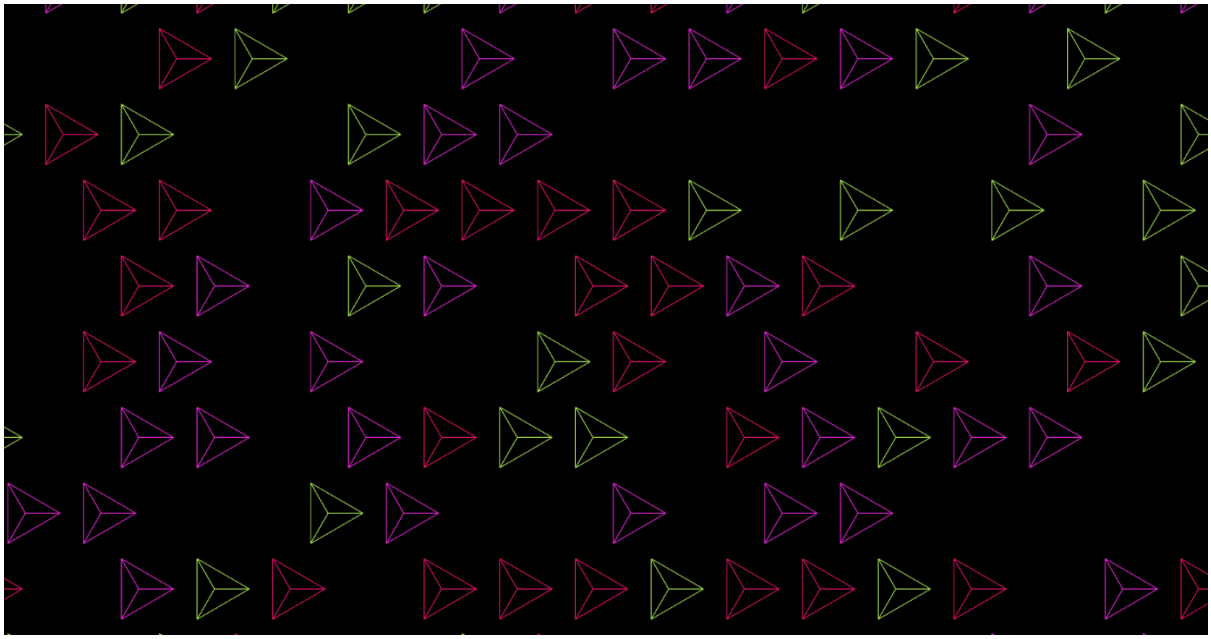

Figure C.5 – Shape Grid (Triangle, wireframe)

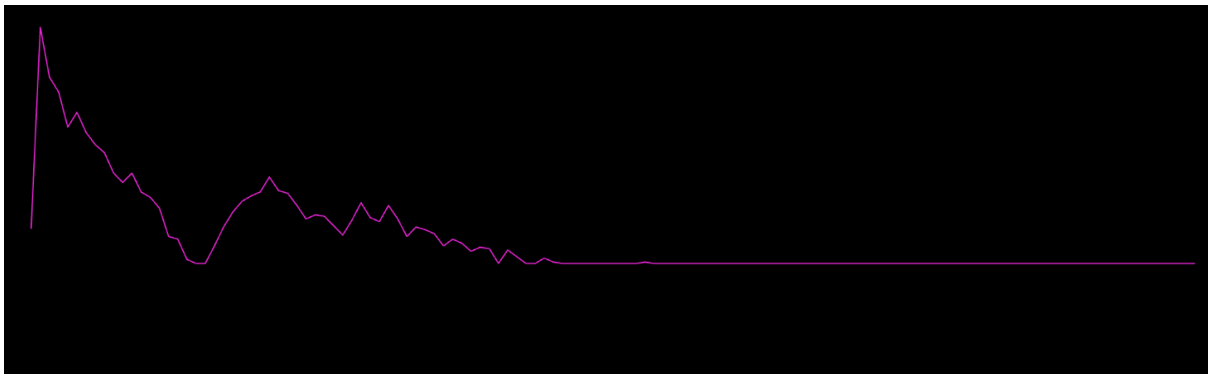

Figure C.6 – Waveform (Solid)

Figure C.7 – Text Visual

Figure C.8 Layered visual

Appendix D – *Diagrams*

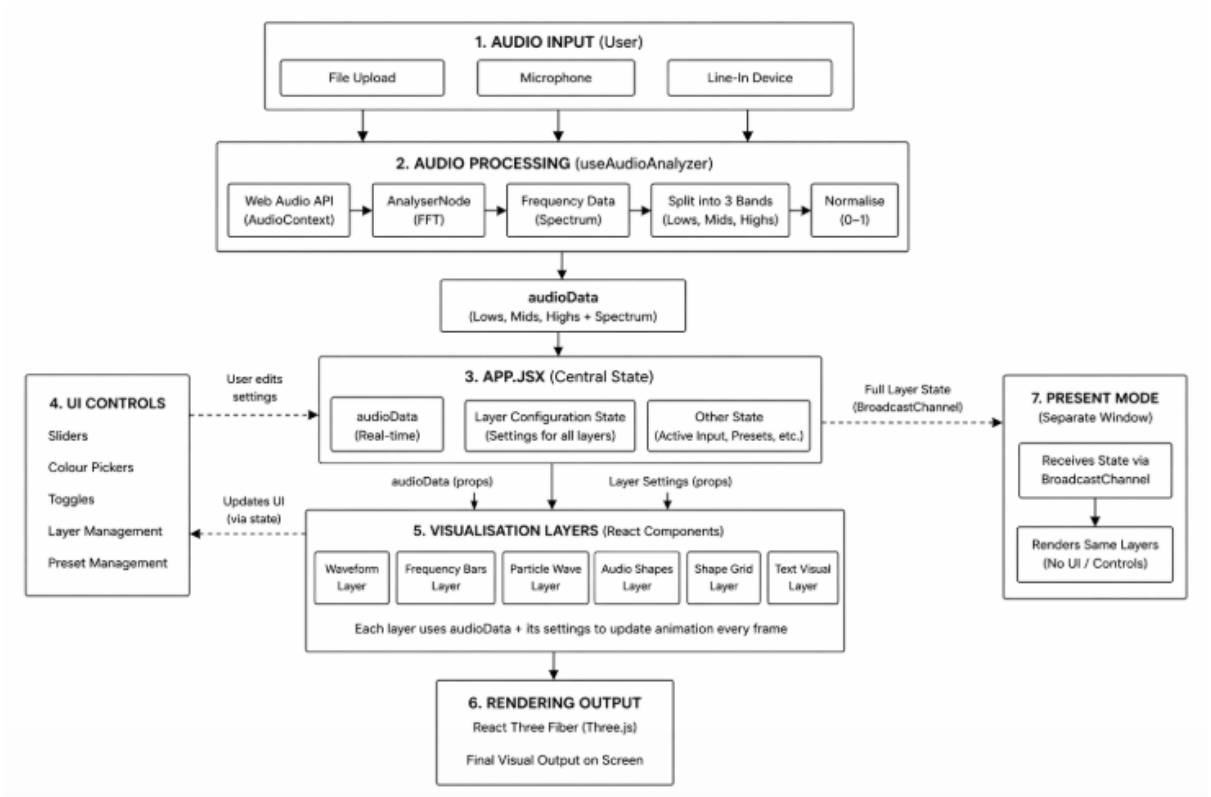

Figure D.1 – Data Flow

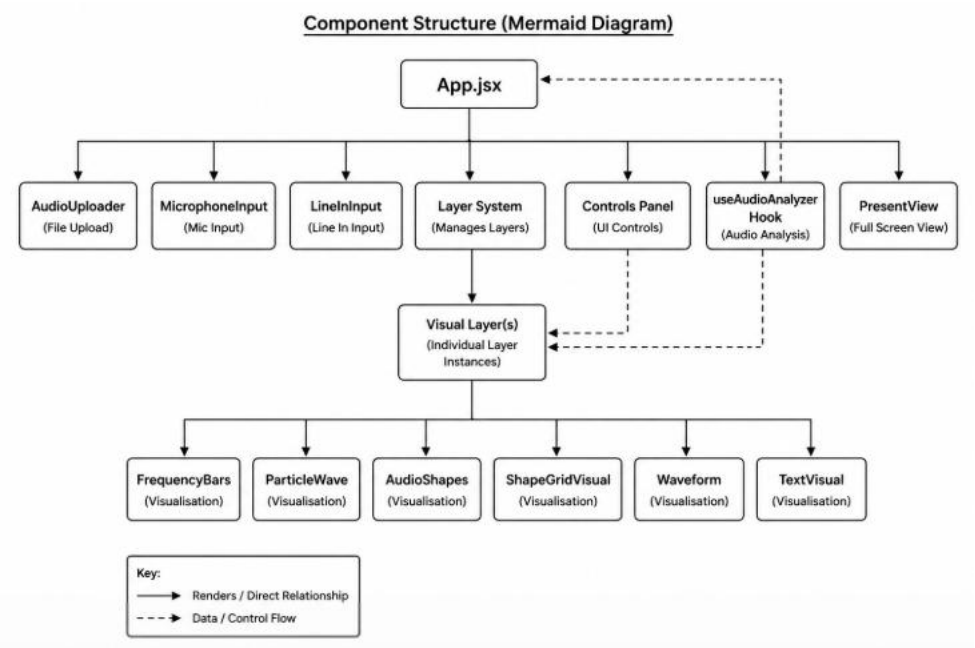

Figure D.2 – Component Structure

Appendix E – UI Screenshots

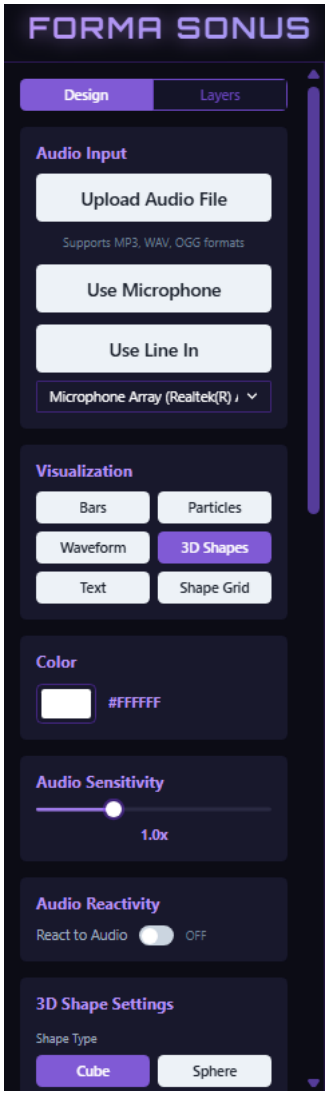

Figure E.1 – Control Panel

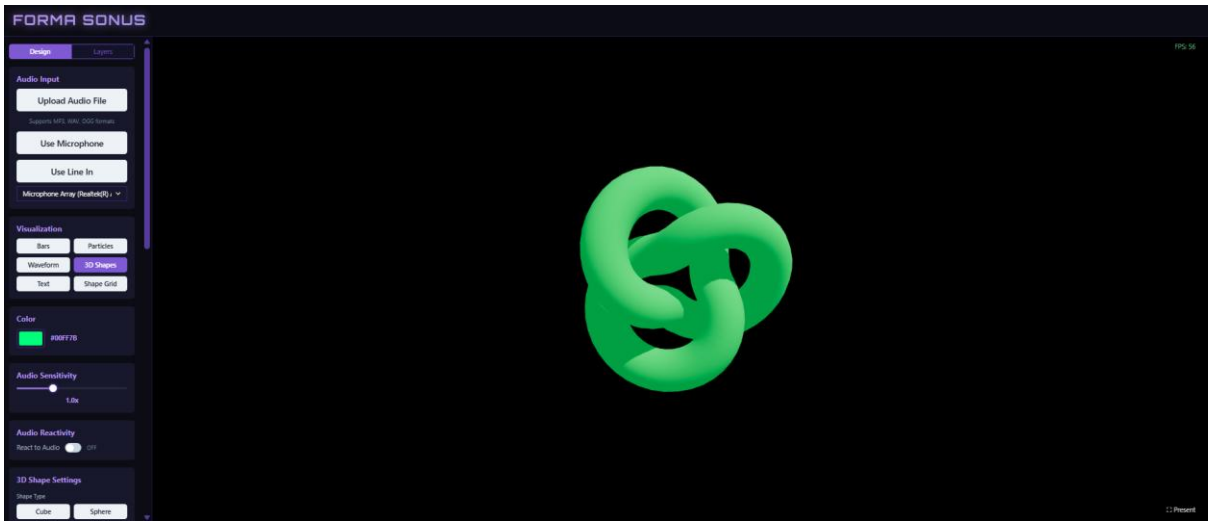

Figure E.2 – UI