Graduation thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Toegepaste Informatica

INTERACTIVE MUSIC VISUALISATIONS WITH P5.JS AND REACTIVE PROGRAMMING

Hanne Sips

Academic year 2022–2023

Promotor: Prof. Dr. Beat Signer
Supervisor: Jorge Isaac Valadez Mora

Sciences and Bio-Engineering Sciences

Acknowledgements

I would like to express my sincere gratitude to Prof. Dr. Beat Signer and supervisor Jorge Isaac Valadez Mora for providing me with the opportunity to delve into my proposed research topic, while granting me the autonomy and freedom to shape its scope. A special thank you goes to Isaac for his invaluable guidance during the process of my thesis. I am also grateful to my boyfriend and study buddy Jeroen for his unwavering enthusiasm about my project and web application, as well as his mental support. My thanks also extend to my friends and colleagues who generously dedicated their time to test the tool, thereby enriching this work with their insights.

Abstract

This study explores the fusion of p5.js and reactive programming to create interactive music visualisations. A user-friendly web application is developed, combining p5.js and RxJs to enable real-time visualisations with interactivity. The study demonstrates the synergy between these technologies, allowing for rapid iteration and easy creation of visuals. The integration of reactive programming enhances responsiveness and flexibility, although performance limitations and learning curves are noted. Overall, this research provides a foundation for newcomers to delve into the world of interactive music visualisations through a powerful, versatile and accessible approach.

Chapter 1

Introduction

1.1 The art of data visualisation

Data visualisation brings data to life. It has proven its value in identifying trends, facilitating decision-making, and presenting information in an understandable way to diverse audiences. A first and well-known example is the NASA global temperature anomaly animation (Figure 1.1) ([1]). It shows monthly global temperature anomalies (changes from an average) between the years 1880 and 2022 in degrees Fahrenheit, where whites and blues indicate cooler temperatures, while oranges and reds show warmer temperatures. This animation serves as a visual representation of the ongoing climate change and its impact on global temperatures. It helps viewers understand the increasing trends and raise awareness about climate change and its consequences.

Figure 1.1: NASA global temperature anomaly animation

Other examples of data visualisation can be found in the geographical context. A Strava heatmap, shown in Figure 1.2, highlights the most popular biking routes on the map. Figure 4 shows a project created by Aaron Koblin, called "Flight patterns": This visualisation interprets flight data in North America and animates them on a map, uncovering human behavioural patterns over time [2].

In the visualisations shown in Figure 1.1, 1.2 and 1.3, the reader with an eye for art can already see an additional potential that goes beyond the boundaries of presenting trends and statistics: data visualisations can lead to true digital art pieces. More than ever, data visualisation has found its way into the art world. It is now part of the discipline called creative coding, which involves coding primarily for an expressive purpose rather than a functional one [3]. Creative coding and

Figure 1.2: Strava heatmap

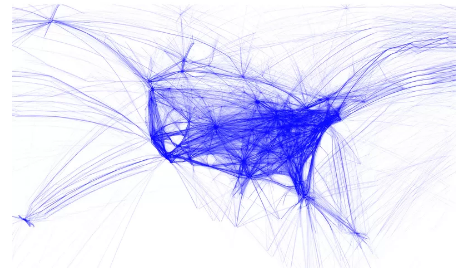

Figure 1.3: Aaron Koblin, Flight Patterns Computer Application (2008)

data visualisation come together in a domain called **data art**. According to Luke Treder, data art "often involves the use of algorithms, software, and technology to turn data into something that can be seen, such as a graph, chart, or animation. The goal is to convey emotions to the audience by sharing insights, patterns, or stories hidden within the data in an accessible and creative way." ([2]) Data artists can build art installations on any kind of input data, like static

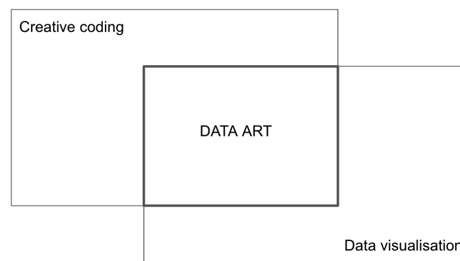

Figure 1.4: Data art, where creative coding and data visualisation come together

data sets, input streams coming from IoT, or real time user interactions ([4]). A captivating example of a data artwork is the Bosphorus Data Sculpture created by Refik Anadol. This unique piece showcases the fusion of real time data flows and artistic expression. Through this sculpture, Anadol transforms complex data related to the Bosphorus Strait water movements in Turkey into a tangible and visually striking form. This work demonstrates how artistic creativity can be intertwined with information visualisation. A snapshot of this interactive artwork is shown in Figure 1.5. ([5]

This thesis zooms in on the visualisation of a specific kind of input data: music. Music can be thought of as a stream of data information over time which, just like other data streams, can be processed and transformed into visualisations. Visuals in the domain of music offer exciting possibilities for enhancing musical or artistic performances by integrating moving animations and captivating visuals that interact with sound or musically instruments. These visual elements serve to augment the auditory experience, creating a multi-sensory spectacle. Moreover, music visualisation extends beyond live performances. Artists have been leveraging the power of visualisation to create digital art pieces like music video clips and immersive audio-visual installations.

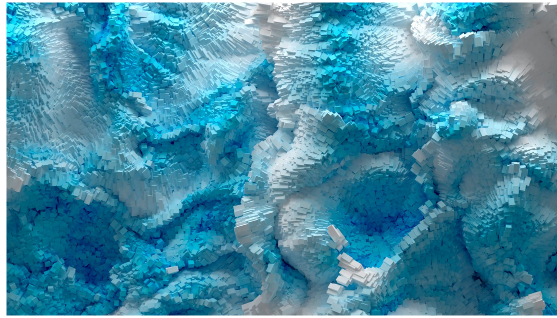

Figure 1.5: the Bosphorus Data Sculpture - Refik Anadol

By synchronising visual elements with audio signals, artists can take their audience on a visual journey that complements and enhances the sonic experience. A snapshot of an example of a digital art animation, made by Memo Akten, is shown in Figure 1.6. It is an animation of a harmonic motion of a symmetric figure that produces sound as if it were a digital instrument. [6]

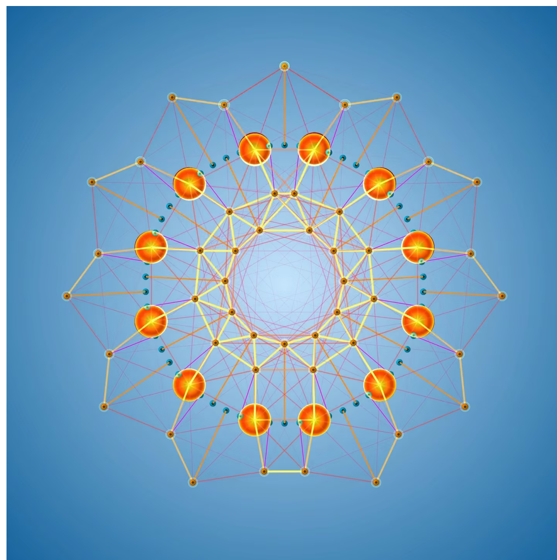

Figure 1.6: Memo Akten - Simple harmonic motion

An example of an interactive art installation is the installation "Flux" of a group of french artists named Scale Collective (Figure 1.7). It is build from 48 beams of light that are each individually motorised and controlled by musical input and by input of the audience through an interface, allowing for a synchronised performance of twisting and coiling patterns. [7] In summary, this thesis explores the intersection of data visualisation and music. The next section delves into the state of the art in music visualisation tools, presenting an overview of the existing methodologies, tools, and technologies available to creators.

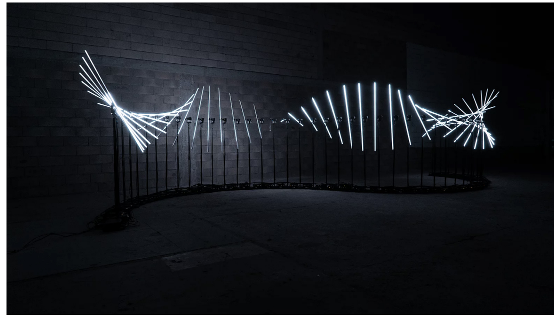

Figure 1.7: Collectif Scale: Flux

1.2 Music visualisation tools

Although real time music visualisation is a rather modern development, there is a large and very diverse range of tools and technologies available, developed by programmers and creators all over the world. These tools serve both non-technical artists and artists with a technical background. For non-technical artists, there are numerous drag and drop tools and drawing applications that facilitate the creation of visual artwork and interactive animations. These user-friendly tools offer intuitive interfaces that enable artists to express their creativity without requiring in-depth programming knowledge. Examples are Magic Music Visuals [1], VZX Music Visualizer [2], MilkDrop [3], Synesthesia [4] and many more.

Creators with technical knowledge and a programming background can find music visualisation technologies in multimedia programming languages, libraries, and packages. Compared to the visualisation tools with prebuilt visuals and effects discussed above, these more technical tools provide a deeper level of control for the creator and make it possible to create visuals and art installations on music data flows from scratch. In what's next, we dive deeper into three kind of technologies: visual programming languages, multimedia specific programming languages and general purpose programming languages.

A visual programming language (VPL) is a programming language that allows users to create programs or applications by manipulating visual elements or graphical representations instead of writing traditional textual code. It is a common technology in music processing applications, as it allows users to process and transform audio and MIDI streams, enabling the creation of synthesizers, visualisers or other sound and visual artworks. By manipulating incoming data streams, users can generate modified audio streams, such as equalisers or filters, or process and transform the streams to visualisation output. The flow of data is often represented visually in a graph structure, making it easier to understand and manipulate. An example of how a VPL looks like, is given in Figure 1.8. [8, 9]

Max MSP [5] and Pure Data [6] are twe VPL's that are both widely used for creating interactive multimedia applications (more on this in Chapter 2). Because of the visual nature of these languages, and the ease of creating new processes in the UI, VPL's are more intuitive and less

[1] https://magicmusicvisuals.com
[2] https://www.vzx-visualizer.com/
[3] https://www.geisswerks.com/milkdrop/
[4] https://synesthesia.live/
[5] https://cycling74.com/products/max
[6] https://puredata.info/

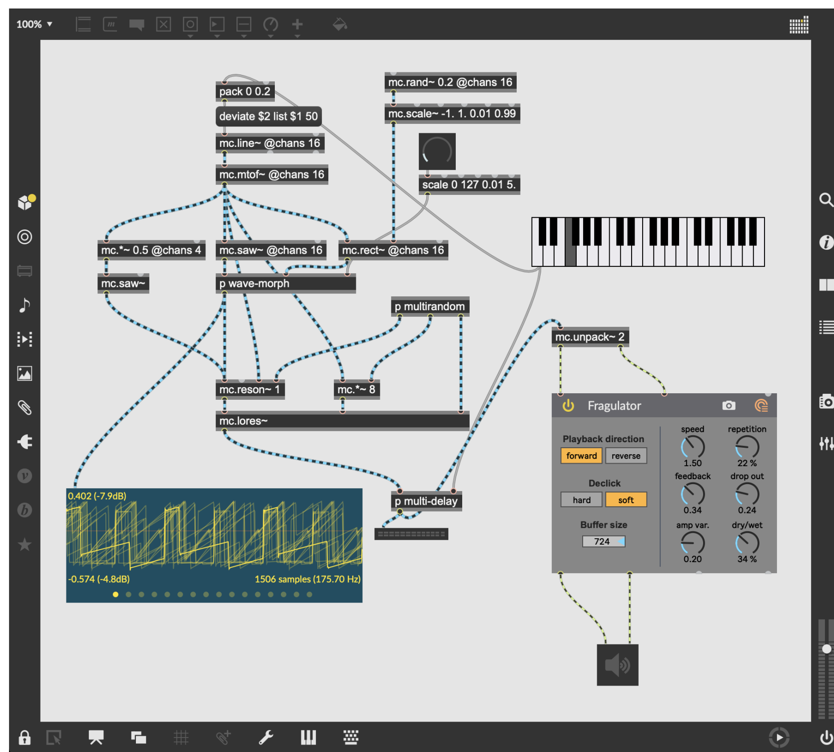

Figure 1.8: Visual Programming Language: Max MSP

technical than textual programming languages. On the other hand, there is a learning phase for technical users to getting used to a new programming style and getting to know the building blocks and features of the language. [10]

Next to VPL's, many text-based programming languages are developed with the goal to facilitate the creation of interactive multimedia applications. Most of them are heavily used in the development of synthesizers, and are also applicable in the creation of dynamic visualisations of sound. A first example is CSound [7], which is an open source, powerful language for creation and modification of music and sound effects in real-time. A second example is SuperCollider [8], an open-source programming language and environment for real-time audio synthesis and algorithmic composition. It is commonly used for creating interactive music and sound design applications. Both CSound and SuperCollider can also be used for creating real-time visualisations of a music data flow. [11]

In addition to languages and tools dedicated to multimedia purposes, more general purpose text-based programming languages like Python, C++, and JavaScript offer packages and libraries that empower users to create sound applications, visualisations and animations. JavaScript, in particular, provides a large variety of packages, including popular ones like p5.js [9] and Three.js [10]. These libraries enable the creation of visualisers and interactive multimedia applications in the browser. [3, 12]

The availability of these diverse technologies offers artists and creators the freedom to choose the tools that best suit their needs, skill level, and artistic vision. Each technology has its own target audience and varying levels of technical complexity and extensibility. In the following sections, we will delve into these solutions, with the main focus on Javascript and it's p5.js library. Further, we will investigate how we can combine p5.js with the principles of a visual programming language to create interactivity of the visual on music data streams. We do this by zooming in on the use of the reactive programming paradigm and find a way to visualise the flow in a graph structure.

1.3 Problem statement and contribution

As stated in the overview above, that only barely touches the surface of all available technologies in the domain of multimedia tools and packages, it is clear that a developer has many options when starting creating music visuals and live performance installations. This variety of tools can be overwhelming to start off. Does the creator want to go in technical detail? Should the creator start learning a new, specified programming language or stick to the more general, widely known programming languages that he/she might already have experience with? If so, which packages suit best the requirements of the project? Is it worth trying a Visual Programming Language in this context, or will it lead to less flexibility and customisability?

Keeping in mind the learning time and costs of getting a true feel for a new tool, technology or language structure (like VPL), there is something to say for choosing a programming language that the creator is already trusted with. Confidence and experience in the programming language can keep the focus of the developer on the creative aspect of the process. Still, as nowadays

[7] https://csound.com/
[8] https://supercollider.github.io/
[9] https://p5js.org/
[10] https://threejs.org/

practically all programming languages have packages available for multimedia programming and audio / midi data stream processing, a choice needs to be made between different languages and libraries.
In this thesis, the possibilities and restrictions of Javascript in the domain of music visualisation are investigated. With the advancement of web browsers, the capabilities for creating dynamic musical content have significantly improved over the past three years. Nowadays Javascript is the most recognised and widely used web technology over the world and is easily runnable by users, regardless of the browser or operating system. JavaScript-based applications running in browsers now offer high performance, portability across various platforms, and long-term viability due to standardisation. The ubiquity of web browsers on desktop and mobile devices makes them widely distributed, positioning them as a "write once, run anywhere" solution for developing musical interfaces ([3, 12]).

This thesis aims to undertake a detailed investigation of the possibilities of JavaScript in the realm of creating digital artwork. Focusing on real time music interactivity of visuals, the combination of Javascript p5 and reactive programming is explored and evaluated. Can reactive programming help in creating interactive music visualisers in p5? This exploration is done through analysis, experimentation, and practical demonstrations.

The research questions of this thesis can be stated as follows:

- An exploration of Javascript p5.js: To what extent is Javascript p5.js suitable for creating digital visuals and animations?
- Enhancing interactivity with music data flows: Can the combination of Javascript p5.js and reactive programming improve the process of creating interactive visuals that respond to music data streams?

1.4 Methodology

Research method

The objective of this research is to explore the use of JavaScript in the context of creative coding, specifically in the realm of visualisations interacting with musical data streams like audio signals and MIDI signals. This research adopts a research through design approach. The primary focus is on the development and evaluation of a web browser tool for creating real time visualisations on music. The design process involves iterative cycles of prototyping, implementation, and refinement.

Functionality

The research aims to develop a prototype web browser tool that facilitates the creation of visualisations in JavaScript, utilising the p5 library. The tool allows developers to connect parameters used in the Javascript code of the visual, to user-configurable and fully customisable inputs that are programmed via the reactive programming paradigm (RxJs). The input stream can be an audio stream recorded by a microphone, a stream of key presses on your laptop keyboard, MIDI streams coming from a synthesizer, a timer event, or any other input that the developer can come up with, as long as it is programmable in Javascript and RxJs. The tool makes it possible to configure and combine multiple input streams, allowing developers to experiment with real-time testing of the effect of parameter value changes on the visual output. By making the connection

between input and visual output easily programmable and real time, it enables developers to concentrate on the creative aspects of visualisation creation.

Technical details

The visualisation tool combines different technologies. The user interface is built using the framework React, that facilitates a modular and flexible tool design. The underlying functionality consist of two parts: the creation and real time testing of visuals, and the configuration of inputs. The creation and real time testing of visuals is configured using the library p5.js, that provides powerful functionalities for generating animated visuals, and creating both 2D and 3D graphics in a HTML canvas element. For the input configuration, the reactive programming library RxJs is explored. This library makes it possible to create, combine and manipulate data streams and connect actions to it (like modifying parameters in the visual code). Inside of the configuration of an input, the Web Audio API or any other built-in Javascript functionality can be used to create fully customisable input streams.The input configuration is made visible in a graph structure with the help of the GOjs library.

Evaluation and Testing

Once the visualisation tool is developed, it is used to provide answers to our research questions: how can p5.js be used to create animated visuals, and how can reactive programming principles help to make these visuals interact with music? This is investigated in the creation process of three demo art works, using different input interactions and visual concepts:

- Demo 1: Visualisation reacting in real time on a midi input stream coming from a connected synthesizer
- Demo 2: Visualisation reacting on the live input of a microphone

During the development of these demo artworks, the use of p5 is discussed, and the influence of reactive programming in interactivity of music data flows is investigated. Different aspects are taken into account, such as: ease of start-up, testability of the visual, ease of connecting the input stream with the visual output, flexibility of tuning the visualisation based on differing parameter values and coding time. In a second phase, the application is tested by a group of test people with different technical background and experience. In a survey and an interview of their findings, the functionalities and technologies of the web application are evaluated in it's strengths and shortcomings.

Conclusion and Implications

The research will conclude with a summary of the key findings and their implications for the field of creative coding and visualisations on music with the technologies Javascript p5 and reactive programming. The conclusion will also discuss the limitations of the study and provide recommendations for further research and improvements to the web browser tool.

1.4.1 Thesis outline

In the following sections, a background in the domain of musical data streams and music visualisation is given. Next, an overview of the work and research done in this domain is provided. With this context, the web browser solution is explained extensively, after which it is evaluated,

tested and demoed. Finally, a conclusion is made on how Javascript p5 can be used as a technology to create music visualisations, and how reactive programming can facilitate the addition of interactivity.

Chapter 2

Background

2.1 Music as data stream

To understand how music data can be transformed to visuals, a basic knowledge of the digital representation of music data is required. The two most common digital formats to represent music tracks are audio and MIDI. Generally speaking, audio is used for capturing live performances or pre-recorded sound, while MIDI is used for composing, sequencing, and controlling electronic music devices. In what's next, both of them are discussed in technical detail.

2.1.1 Audio input stream

Audio data refers to the digital representation of sound waves. It captures the actual audio signals produced by musical instruments, voices, or any other sound source. Audio data is typically stored as a sequence of samples, where each sample represents the amplitude of the sound wave at a specific point in time. These samples are captured at a certain sampling rate, usually measured in Hertz (Hz). Audio data is widely used in formats such as WAV, MP3, AAC, and others. It can be played back on various devices, including speakers, headphones, or computer systems, to produce audible sound.

Audio data is typically represented in either the time domain or the frequency domain. Time Domain: In the time domain, audio data represents sound as a waveform. The waveform shows how the sound pressure level changes over time. It is a one-dimensional representation where time is plotted on the x-axis and the amplitude (loudness) of the sound is plotted on the y-axis. This format directly captures the variations in sound over time and is commonly used for tasks such as audio recording, playback, and editing.

Frequency Domain: The frequency domain representation of audio data provides information about the different frequencies present in a sound. It is obtained through a mathematical transformation called the Fourier transform. The Fourier transform decomposes the time domain waveform into its constituent frequencies. In the frequency domain, audio data is represented as a spectrum, with frequency on the x-axis and amplitude on the y-axis. This representation is useful for analysing and manipulating the individual frequency components of a sound. It is commonly used in tasks such as audio equalisation, filtering, and spectral analysis.

The time domain and frequency domain representations of audio data are closely related. The Fourier transform allows us to convert between the two domains, enabling various audio processing techniques. [13, 14]

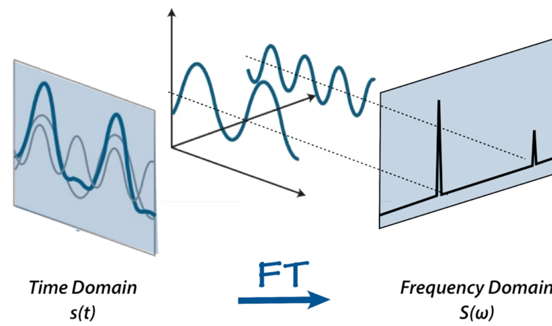

Figure 2.1: audio signal in time domain and frequency domain

2.1.2 MIDI input stream

MIDI (Musical Instrument Digital Interface) data is a standard protocol used to communicate musical information between electronic musical instruments, computers, and other devices. It represents music in a different way compared to audio data. MIDI data does not directly capture the sound itself, but rather represents the instructions or commands that control musical parameters. MIDI data consists of a series of messages or events. Each message typically includes information about aspects such as note on/off events, pitch, duration, velocity (how hard a key was pressed), and various control parameters (e.g., modulation, expression, sustain). These messages are sent and received between MIDI-enabled devices to generate and control sound. ([15]) An example of a MIDI message is given in Figure 2.1 . ([16])

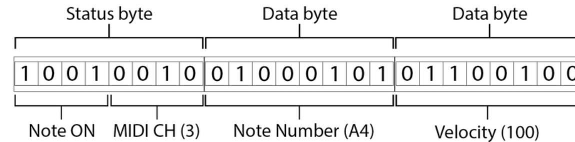

Figure 2.2: format of a midi signal

MIDI data is highly versatile and can be edited, manipulated, and played back on different MIDI-compatible devices. For example, a MIDI keyboard can send note on/off messages to trigger the corresponding sounds in a software synthesizer or a hardware MIDI sound module. MIDI data is often used for tasks such as composing music, recording MIDI performances, and controlling virtual instruments. It's important to note that MIDI data does not contain actual audio wave forms, so it requires a MIDI-compatible device or software to generate sound based on the instructions it provides.

2.1.3 Audio and midi processing

Generally speaking, audio is used for capturing live performances or pre-recorded sound, while MIDI is used for composing, sequencing, and controlling electronic music devices. Keeping in mind the difference between audio and midi signals, let's look at a common musical hardware setup and it's associated data flows in Figure 2.2: The interface device handles a combination of audio signals and midi signals, by connecting three input devices, microphone, midi controller

keyboard and computer, to three output devices, monitor loudspeakers, monitor headphones and computer (can be both input and output device). ([17])

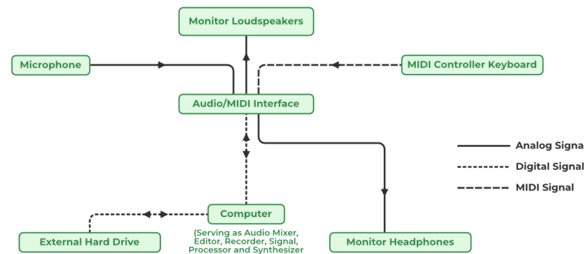

Figure 2.3: audio data flow versus midi data flow

Let it be clear that, although both audio and midi data represent music tracks, different technology and processing is needed to transform it into visuals. ... [discuss the processes further]. In live performances, also a combination of audio and MIDI is possible. In this thesis, visualisation of both audio and MIDI device input is explored.

2.2 Reactive programming

Reactive programming is a programming paradigm that focuses on asynchronous data streams and the propagation of changes. It is primarily used to handle and respond to event-driven and real-time systems, where data is constantly changing and events are continuously generated. A list of the core concepts of reactive programming:

- **Observables** represent a source of data or events that can be observed over time. Observables can emit values, error messages, or completion signals.
- **Subscribers** are entities that subscribe to observables in order to receive and react to the emitted values. Subscribers define functions or callbacks that are invoked when new values are emitted.
- **Operators** or methods transform, filter, combine, or manipulate the emitted values from observables. These operators enable powerful data processing and manipulation capabilities.
- **Event-driven**: Reactive programming is particularly well-suited for event-driven systems where events occur asynchronously and need to be handled in real-time. It provides a declarative way to express the behavior of the system based on the incoming events.
- Reactive programming allows for efficient handling of **asynchronous operations**. It provides mechanisms to handle asynchronous tasks, such as network requests or user input, without blocking the execution flow.

In the example below, an observable emitting click events is asynchronously processed by three sequential operators: buffer, map and filter. The output is again an observable that can be subscribed to by a subscriber. The subscriber can execute a callback function on every emitted value, for example printing a string in the console.

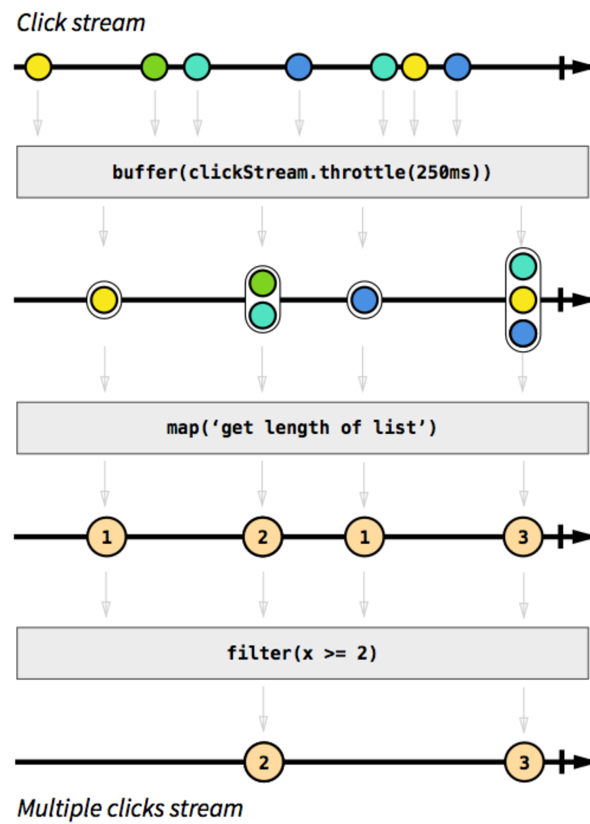

Figure 2.4: Reactive programming example

Seeing the benefits of reactive programming, it is commonly used in the domain of web and mobile development, for creating interactive and responsive user interfaces that handle dynamically incoming data streams and user interactions. Also for IoT applications, reactive programming is well-suited to handle and efficiently process events and changes in the IoT environment. Also in the contexts like gaming, microservices and distributed systems, reactive programming has proven it's value.
What about the context of music data flows and processing? Some papers explored the possibilities and advantages of reactive programming in manipulating audio signals or performing live music coding ([11], [18]). They state that "reactive programming is a way of modelling stateful streams of data updates, e.g. from hardware musical instruments like MIDI control surfaces, in a way that allows the programmer to write pure functional code that operates over them". In the concept of transforming music streams to visuals, reactive programming principles can help in introducing responsiveness of the visual to music data flows efficiently. In reactive programming terms, subscribers listening to observables emitting music signals, can process these signals asynchronously and transform them into changes in the visual. The technical implementations of such a setup are described in detail in the Solution section.

2.3 Visual Programming Language

A visual programming language (VPL) is a type of programming language that uses visual elements, such as graphics and diagrams, to create and execute program code. Instead of writing code using text-based languages, VPL allows developers to create programs by dragging and dropping blocks or symbols that represent different functions or actions. VPL's offer a number of benefits such as ease of use, also for non-technical profiles, rapid prototyping and intuitive syntax. ([19])

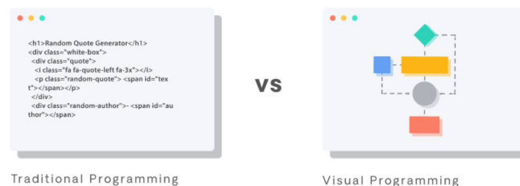

Figure 2.5: Text based programming versus visual programming

In the context of music and sound data processing, two important VPL's are Max MSP and Pure Data. **Max MSP** is a commercial visual programming language developed by Cycling '74. It is widely used in multiple fields of music and sound design, and it offers a wide range of modules and objects that can be connected together to create complex interactive systems. Max/MSP is known for its versatility and extensive library of pre-built objects, making it suitable for a broad range of applications beyond just music. It has a strong presence in the professional audio and multimedia industry and is widely used in areas such as live performances, installations, and interactive media. The Max MSP programming language allows users to build and manipulate music data flows, by creating building blocks (sources, functions and outputs) and connecting them to each other. Jitter, an extention of Max MSP specialized in real-time video processing, makes it possible to turn data flows into animated visualisations. ([20]

On the other hand, **Pure Data (Pd)** is an open-source alternative to Max/MSP that follows a similar visual programming paradigm. It was initially developed by Miller Puckette, the creator of Max, and is freely available for anyone to use and modify. Pure Data focuses primarily on real-time processing of music and audio data flows, and it has a strong community of users within the electronic music and digital art domains. While Pure Data may have a slightly narrower focus compared to Max/MSP, it still offers a wide range of functionality and can be extended through the creation of custom objects and patches. ([21]

Overall, visual programming is a powerful tool for multimedia purposes, as it enables artists, designers, and creative individuals to explore and experiment with audio and visual elements in an intuitive and visual manner. Moreover, visual programming languages provide a natural environment for expressing reactive programming concepts. Their graphical representations and visual connections between blocks make it easier to understand and work with reactive behaviour. Nodes representing music data streams and events, and operations can be connected to propagate changes and trigger updates throughout the program. This integration of visual and reactive programming enables the creation of dynamic and responsive applications with ease.

2.4 Visualisation of music

In the early days of music visualisation, pre-built and automatically generated visuals were available to everyone through applications like iTunes or Windows Media Player. Many people will recognise the colourful shapes and movements of the spirals in Figure 2.3. These visuals were synchronised to the frequencies and loudness of the song, creating a mesmerising audio-visual experience.

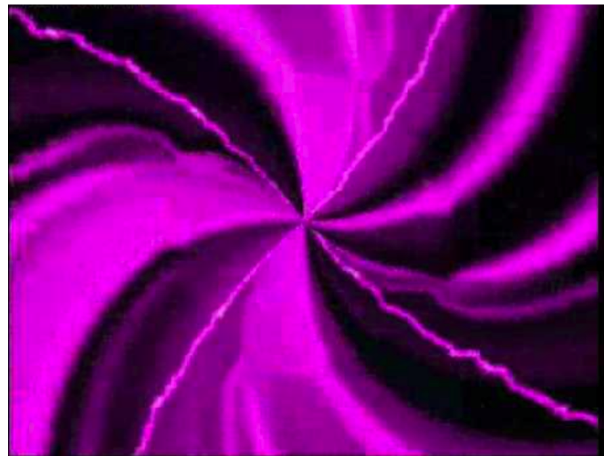

Figure 2.6: Early day music visualisations

Over the years, technology in the field of music visualisation has rapidly evolved, especially with the rise of electronic music. Interactive visuals and lightning have become a standard in almost every large club, and DJs and artists now incorporate moving animations that enhance their musical performances. These interactive visuals react in real-time to the music being played, creating an immersive and dynamic experience for the audience.

Today, visualising music has become an art form of its own, with various software and hardware tools available for artists and designers to create captivating and personalised visuals that

complement the music. From real-time projections on stage to interactive installations at music festivals, music visualisation has expanded its boundaries and possibilities, blurring the lines between auditory and visual artistic expressions.

Chapter 3

Related Work

Many research has been done about the emotional effects and impact of combining music with visual stimuli. A paper of Xiangnuo and Jialu ([22]) states that "digital art works under multi-modality provide not only sensory experience, but also emotional value to the audience through digital technology and multi-sensory interaction, thus forming a unique artistic charm".
Another paper explores the value of multimodal stimulation in the context of musical experience and learning. It indicates that the multimodal nature of musical involvement and expression is an important aspect of embodied music cognition. The paper emphasises the fact that visual and auditory perception do strongly interact. Regarding music education purposes, interactive music systems can address current issues regarding motivation and creativity in music education, and help in the creative use of musical parameters ([23, 24]). Other papers investigate the power of music visualisations for people with one or multiple disabilities. It states that by combining music with, for example, tactile stimulation, movement, or visuals, meaning-making processes in music of these people was stimulated, helping them to understand the internal structures and expressive qualities of music. ([25])
Other research investigates the neural and stress-reducing effects of music in combination with visual art. It finds that multimodal (music + visual art) aesthetic experience has a stronger effect than a single modal aesthetic experience (music/ visual art). ([26, 27]

The rapid progress of science and technology has led to great progress and innovation in the forms of digital art ([22]). The transformation of audio signals to visuals is investigated in previous research. One paper introduces an audio analysis plugin for real time processing, allowing users to select a custom set of audio analyses to be performed in real-time. ([28]) Other research is done about a general transformation of audio signals to visuals. It investigates methods for interactively mapping audio features (numerical data representing signal and perceptual attributes of sound/music obtained computationally) to visual objects to facilitate music visualisation. Further, it gives a concise introduction to the theory of how information can be derived from sound signals and presents examples of how that information can be visualised using Processing. ([29])

Reactive programming in the context of music data flows is investigated in a paper that presents an alternative approach for callbacks and explicit state programming for creating synthesizers in the SuperCollider audio synthesis environment. In this paper, functional reactive programming (FRP) is used to define the control logic of the instrument, where inputs are taken from musical controllers, mobile apps or graphical user interface (GUI) widgets.

Several research is done regarding the use of visual programming languages in data flow process-

ing and more in particular in the context of music applications. A first paper investigates the use of VPL's to process data flows in a context of IoT (Paper [4]). In the domain of music processing, papers [8] and [9] confirmed in their research that the visual nature of VPL's is suitable for music data flow processing, and that these languages remove hurdles between artists and their ideas, enhancing the speed of the creation process. Further, it compares multiple visual programming languages, like OpenMusic, Max MSP and Pure Data, in a survey, discussing the pro's and con's of each language. Another paper ([10]) investigates the value of Max in processing digital music signals, with the goal to "promote the communications in art and other fields, and to some extent inspire the transformation of relevant art education". Next to Max, OpenMusic is investigated in a paper ([30] where the link between visual programming and reactive programming is explored. The paper presents a fairly simple and elegant formalism for the integration of reactive processes in the visual programming language OM. The visual programming language Sonnet in combination of Imager is investigated in the context of creating dynamic visualisations ([31]), resulting in a system that is capable of generating a wide variety of dynamic visuals. In the paper [32], a music visualisation system is described that creates images interacting with both midi and audio data. The system is designed in a way that makes the process of artist/scientist collaboration as easy as possible, with the use of visual programming to develop system components whenever possible. Technologies like Max/MSP, Jitter, and Virtools allow creators without training in traditional computer programming to participate in the development process with greater ease.

Next to VPL's, many text-based programming languages are developed to facilitate the creation of interactive multimedia applications. Processing, the programming language on which the Javascript p5 library is inspired, is a widely known and used by artists and designers, for different purposes, from scientific data visualisation to art installations ([33]). Research investigates the use of Processing in the domain of music visualisation and audio signal processing. Further, Processing with the purpose of interactive visualisation of music and user input is investigated in the creation of a music player visualisation that integrates low-level features of music data via shapes and colours with real-time animation and player control interactions ([34]. Another paper investigates Python as a programming language for real time music visualisation, by implementing a Python real time music visualisation method. More specifically, a system is implemented capable of visualising music by analysing the time and frequency information of the digital music and utilising Python sound libraries ([35])

For programmers who are already familiar with Javascript, there is a wide range of libraries available for creating multimedia content and interactive applications. Research has proven the advantages, flexibility and testability of the creation of digital instruments in the web browser, using the Javascript library Gibberish.js and the Web Audio API ([12]). The Javascript p5 library, that brings the simplicity and creative possibilities of Processing to the web, is explored and evaluated. P5 is found easy to get started with, intuitive, easy to read, applicable for a lot of purposes, an excellent choice for experimenting with creative coding, however performance is reaching its limits in professional contexts and large scale projects ([3]).

Chapter 4

Solution

4.1 General concept

In this research, a web browser tool is developed that enables the creation of interactive visuals. With this tool, the user is able to create visuals by coding in Javascript p5.js, and tuning visual parameters in real time by connecting them to user configurable input data streams using the reactive programming paradigm. The tool exists of three integrated components:

- an input configuration component (left): a component where input data streams can be configured, manipulated and processed. Example input data streams are audio input, MIDI signals and key presses.
- a Javascript p5 code playground (center), where the developer can create an image or animation through Javascript p5 code.
- and an output canvas (right) displaying the visual.

These are shown in Figure 4.1. Each of these components can be clicked and expanded. At the top, two buttons are providing functionality for saving the visual configuration or uploading a previously configured visual into the web tool. In what's next, the functionality and technical implementation of each component is discussed in detail.

4.2 The input configuration component

4.2.1 Functionality

The input configuration component handles the configuration of all possible input data streams and their influence on parameters in the javascript visualisation code. The input selection component is using the RxJs library to configure multiple input data streams and process them to parameter changes. The section is implemented by using a graph structure, containing nodes and edges connecting nodes. The different types of nodes are consistent with the concepts of RxJs (see Background sections and technical implementation for more details on this library):

- **Observable node**: rxjs observable emitting values / events. An observable has a name and a configuration that are both configurable by the developer. They can be changed by clicking on the observable node, so that the name and configuration input fields are

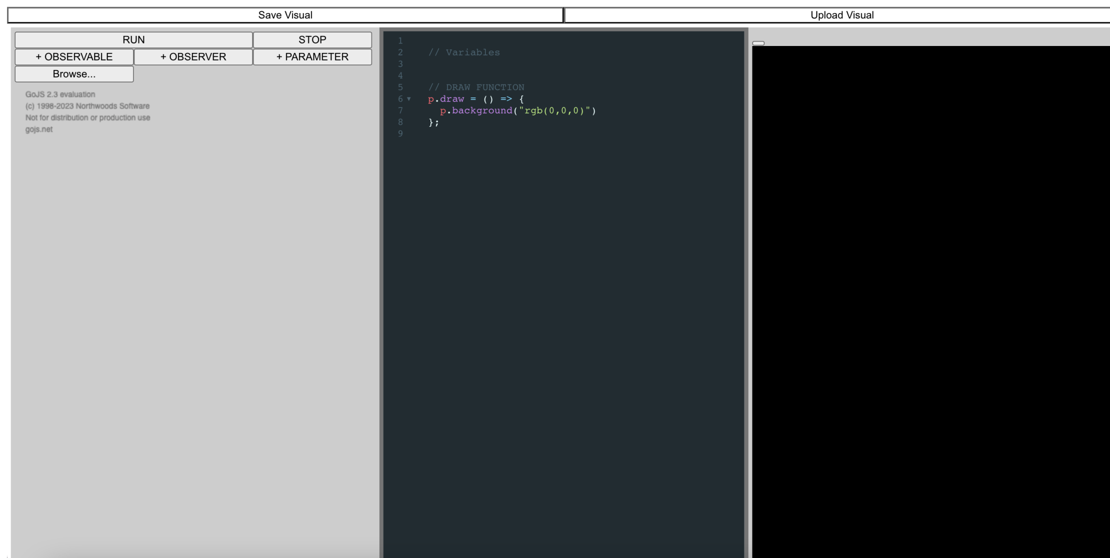

Figure 4.1: tool layout

opened at the bottom of the input selection component. When doing so, the node is coloured in purple to indicate that the configuration of this node is opened and editable. The configuration of the observable node is shown in a javascript editor, that can interpret general Javascript and RxJs code. A valid observable configuration is returning an rxjs observable in the editor. When the observable code is invalid, the node colours red and the error message is added below the editor. To make it more concrete, a screenshot of the observable configuration layout is shown in Figure 4.2.

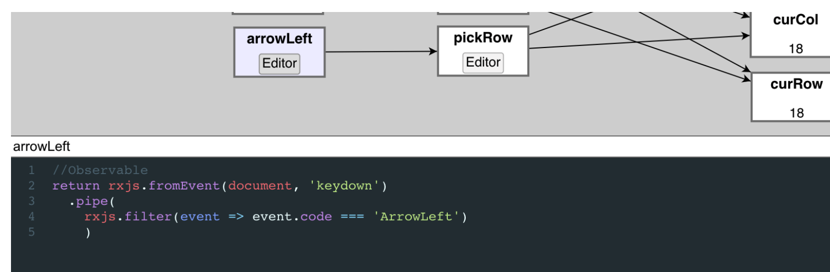

Figure 4.2: Observable node configuration

- **Parameter node**: a parameter that can be used in the visualisation editor A parameter node has a name and a value. The parameter name can be customised by the developer and the value field is always showing the real time value of the parameter.
- **Observer node**: nodes connecting observable nodes with parameter nodes. They are RxJs observers listening to observable(s) and providing a callback function that changes the value of one or more parameters depending on the values emitted by the observable(s). An Observer node has, like an observable node, a name input field and a configuration in a Javascript editor. The node name and configuration can be opened and changed by

clicking on the node, so that the name input field and Javascript editor field are opened at the bottom of the input selection component. The node again colours purple indicating that it can be edited. A valid observable node subscribes an observer callback function to an observable node in the Javascript editor. When the observable code is invalid, the node colours red and the error message is added below the editor. An example of an observer configuration is shown in Figure 4.3.

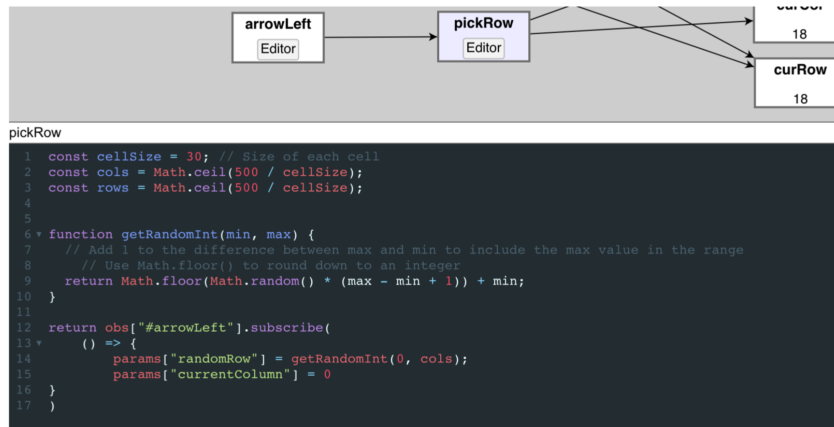

Figure 4.3: Observer node configuration

A full example of an input data stream graph is shown in Figure 4.4 where three different observables are emitting values, while three different observers are reacting to these value emissions by changing the values of four parameters. These parameters can be used in the visual code of the second component. The graph can be customised to the wishes and needs of the developer: new nodes of each type can be added by the three buttons "Observable", "Observer" and "Parameter" at the top, nodes can be removed by clicking on them and pressing the Backspace key, nodes can be moved around the graph to make the graph layout more structured, node names can be changed, the graph can be zoomed in and out or moved entirely to the left or right to zoom in on a specific part of it.

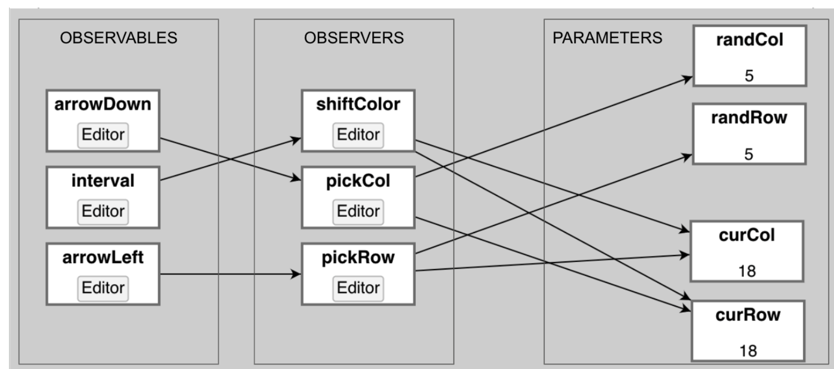

Figure 4.4: Input selection component - example graph

As a help for the developer, some preprogrammed observables are foreseen in the tool. They

can be opened as a node in the graph by clicking on the "Browse" button. Functionalities like MIDI signal processing, audio frequency signal generation, key presses or interval observables are currently present in the list, that can be extended in the future. When the developer has created and configured some observables, observers and parameters working together, the setup can be tested by pressing the "RUN" button. Automatically, links between observables, observers and parameters are calculated and presented by an arrow in the graph. More concretely, observables have outgoing edges to all subscribed observers, and observers have outgoing edges to all parameters that are affected by this observer. From the moment the input component is running, new emitted values are captured and the paths they affect are visually represented by highlighted graph nodes in yellow. through the graph When the STOP button is pressed, all observables quit emitting values and no parameter values are changed anymore.

4.2.2 Technical implementation

- RxJs

The input section uses a combination of reactive programming in RxJs, and a visual graph representation implemented by the GO.js framework. To understand how this component is working technically, let's first dive into a short introduction of the RxJs library. Reactive programming is

a programming paradigm that focuses on incoming data streams and the propagation of changes. It allows you to express the dynamic behaviour of a system in terms of streams of events that can be observed and reacted to. This is explained in more detail in the Background section. RxJS (Reactive Extensions for JavaScript) is a popular library that brings the principles of reactive programming to JavaScript. It provides a set of powerful tools for working with data streams and enables you to easily compose and manipulate them in a declarative manner. At the core of RxJS

is the concept of an **Observable**. An Observable is a representation of a stream of values that can be observed over time. It can emit values, errors, and completion signals. Observables can be created from various sources such as events, callbacks, timers, or even existing data structures. Once you have an Observable, you can apply a wide range of **operators** to transform, filter,

combine, and manipulate the data stream. These operators allow you to perform operations like filtering values, mapping them to different values, merging multiple streams, and much more. RxJS provides a rich set of operators that can be combined in various ways to create complex data flows. An operator has one or more observables as input and has an observable as output. An **observer** is an object or a set of callback functions that can subscribe to an Observable to

receive notifications when values are emitted. An observer typically consists of three optional callback functions: next, error, and complete. The next function is called whenever a new value is emitted by the Observable, the error function is called when an error occurs, and the complete function is called when the Observable has completed emitting values. Observers can subscribe to an Observable using the subscribe method and can unsubscribe to stop receiving further notifications. This immediately explains the concept of a **Subscription**. A Subscription represents the connection between an Observable and an Observer. Subscriptions can be used to control the lifecycle of the observation and to unsubscribe when you're no longer interested in receiving updates.

These four concepts are disposable inside the nodes of the input selection component. An observable node can contain any observable available in RxJs, possibly transformed by one

or several operators (again returning an observable). An observer node can subscribe to one of the observable nodes with a certain callback function. This link between observable and observer created a subscription object. The application provides three dictionary objects to store and access the observable objects, observer objects and parameter objects created in the input diagram. When the **"RUN"** button is clicked, the following steps are run:

- (0) If any previous configuration was already running, it is stopped (more on this later).
- (1) All parameter nodes in the diagram are created and added to a dictionary called params, with the parameter name as their key. Other nodes can now access these parameters by extracting the Parameter objects out of the params dictionary.
- (2) Links between observable nodes, observer nodes, and parameter nodes are identified by conducting a text search within the node editors. As an example, the following code snippet is used to establish a connection between observable and observer nodes:

```
function connectObsvblsToObsvrs() {
  observables.forEach(observable => {
    observable.observers = []
    observers.forEach(observer => {
      if (observer.code.includes(observable.name)) {
        observable.observers = [...observable.observers, observer]
      }
    })
  })
}
```

- (3) All observable nodes are executed and the returning Observables are stored in the obs dictionary with the observable name as their key. Other nodes can now access these observables by extracting the Observable out of the obs dictionary. If an observable editor contains an error, the observable node is highlighted in red and an error message is added below the editor.
- (4) All observer nodes are executed and all configured subscriptions are created and stored in a dictionary called subscriptions. For failing observer code, the observer node is highlighted and an error message is added below the failing observer editor.

After all these steps, the diagram is updated with the newest changes (links, errors) and new parameter values are calculated and shown continuously. For every newly emitted value by an observable, active nodes are highlighted. The continuously updating params dictionary can now be used in the visualisation code in the visual playground component.

When the developer clicks on the **"STOP"** button, or when the RUN button is clicked and a new configuration needs to be set up, several steps need to happen to shut down the current input configuration. 1. the params dictionary is cleared 2. all subscriptions in the subscription dictionary are unsubscribed from their observables. This is needed to avoid that observers keep on listening to observables that no longer exist. After that, the subscriptions dictionary is cleared. 3. all observables are completed so that they no longer emit values. After that, the obs dictionary is cleared.

These steps are necessary to avoid memory leaks in the input component. All RxJs objects need to be shut down explicitly, otherwise event emissions and event listeners will keep on running

and will slow down the application significantly in every new run. The code snippet of the STOP functionality is given below.

```
function stopExecution() {
    params = {}

    // stop subscriptions
    for (const subscription in subscriptions) {
      subscriptions[subscription].unsubscribe()
    }
    subscriptions = {}


    // stop observables
    stopObservable$.next();
    stopObservable$.complete();

    obs = {}
  }
```

- DIAGRAM

The diagram layout in the input component is implemented with the help of the GOJS library. GOJS is a versatile JavaScript library that empowers developers to build interactive and customisable diagrams for web applications. With GOJS, you can create a wide range of diagram types, such as flowcharts, organisation charts, mind maps, UML diagrams, network graphs, and more. It includes support for interactive behaviours like node dragging, link creation, grouping, panning, and zooming. The library also facilitates data binding, allowing seamless integration with external data sources to dynamically update the diagram content.
The input configuration diagram is build as a GOJS GraphLinksModel, that enables creating nodes and linking them with each other. It stores node data and link data in two separate arrays. Each node and link is adaptable by both user interaction (like changing a node name or opening a node editor colouring the node purple) and application logic (like a parameter value changed by an observer callback function or a node highlighting because an observable is emitting a new value). To enhance user experience, GOJS offers smooth animations for changes to the diagram and supports commands and actions for intuitive user interactions. Additionally, it provides an undo/redo mechanism for users to revert changes easily. These functionalities make user interaction with the input diagram smooth and flexible.

4.3 The Javascript playground

4.3.1 Functionality

The visual playground component of the tool is a Javascript code editor in which the developer can create visuals that are run in real time. The editor has a p5 running environment, so all functionality available in p5 can be used to create visuals: shapes, colours, lighting and effects. The visual code can exist of two parts:

- Setup variable declarations and functions, executed only once
- a draw function executed continuously to run the animation

Every time a change in the visual code is detected, the full editor code (part 1 and 2) is rerun. From then on, the draw function is continuously executing in a loop, updating the visual in the output component. When an error is detected while running the visual code, the visualisation is no longer updated and an error message is shown below the editor. From the moment the issue is fixed, the visual appears again in the output component. This makes it easy and hands on for the developer to test and evaluate the visual code. In the Javascript editor, supportive features such as autocompletion, indentation and selection highlights are foreseen to enhance the developing experience (however it can still be improved).

Here is where the input component and the visual playground component come together: if the developer configured some input parameters in the input component, these parameters can be used in the visual code, by accessing the params dictionary:

```
params[name]
```

As the draw function is executing continuously, changes in parameter values are affecting the visual in real time. The visibility of parameters values and changes in real time in the diagram, supports the developer in understanding the movements in the visual. Also,testing the effect of changing parameters to the visual is easy and fast. This keeps the focus of the developer on the creative process, and aesthetic enhancement of the visual.

4.3.2 Technical implementation

To understand how the visual playground editor works technically, a quick introduction is given on how p5 works.
Overall, p5.js provides a beginner-friendly and intuitive way to create visualisations in a canvas using JavaScript. It abstracts away many of the complexities of raw HTML5 canvas programming and provides a simplified and expressive API for creating interactive graphics on the web.

Setup: Every p5.js sketch begins with a setup() function. This function is called once when the sketch starts and is used for initialising settings and variables. You can set the canvas size, choose a background colour, or any other initial configurations you need.

Draw: The draw() function is the heart of a p5.js sketch. It is called continuously in a loop after the setup() function. The draw() function is responsible for updating and rendering the visuals on the canvas. You can think of it as the main animation loop.

Canvas: The canvas element is where your visualisations will be displayed. You can create a canvas by using the createCanvas() function in the setup() function. This function takes parameters for the width and height of the canvas.

Inside the draw() function, you can use various built-in p5.js functions to draw shapes, lines, colours, and images on the canvas. For example, you can use functions like rect(), ellipse(), line(), background(), and fill() to create different visual elements.

In the visualisation tool, p5 is present in the running environment of the Javascript visualisation editor. Moreover, the setup function, that as stated above is specific to the p5 package, is running in the background from the moment that the application is started, and creates the output canvas component in the output section. In other words, the setup function is already created in the application code and does not need to be defined by the user. This makes sure that

the user doesn't need to worry about how the canvas is appearing on the web page. From the moment a valid draw function is typed in the visualisation editor, the coded visual is appearing in the canvas. If the visualisation code is not valid, an error message is printed out at the bottom of the visual playground component.

4.4 The output canvas component

The output canvas is where the first and second component come together: the visual image or animation is run and shown in the canvas, and the influence of changing parameter values is immediately visible in the animation. This makes it possible for the creator to test the visualisation code very fast, and try out parameter changes to create a true art piece. Further, the animation can be opened in full screen mode to make screen recording possible or hide the technical details for a live audience.

4.5 From manual to practice: a first example

Despite a rather simple tool layout, the three components work together in a flexible way and with a lot of possibilities. In what follows, the three components and how they work together, are explained extensively and with the help of an example, the visualisation and animation of a grid of circles (Figure 4.5).

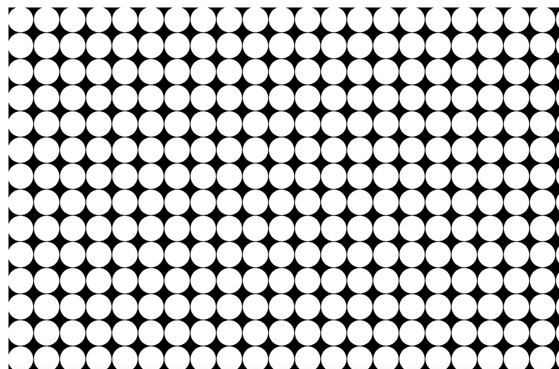

Figure 4.5: first example p5 visual

The following sections explain:

- how this image is created in the tool by the help of the p5 framework
- how the image can be animated by an input stream of key presses on the user's keyboard

4.5.1 Visualisation code

The code that creates the grid of circles is given in the code snippet below. Outside of the draw function, the cell size and initial circle size are declared. Out of the cell size, the number of rows and columns is calculated. The draw function, that is executed continuously, paints the background in black and loops over all rows and columns to draw each circle.

```
// Variables
const cellSize = 50; // Size of each cell
var circleSize = cellSize // initial size of each circle

const cols = Math.ceil(visualWidth / cellSize);
const rows = Math.ceil(visualHeight / cellSize);

// DRAW FUNCTION
p.draw = () => {
  p.background(0);
  p.translate(-visualWidth / 2, -visualHeight / 2);

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const x = j * cellSize + cellSize / 2; // X-coordinate of the circle's center
      const y = i * cellSize + cellSize / 2; // Y-coordinate of the circle's center

      // Draw a circle at the calculated position
      p.circle(x, y, circleSize);
    }
   }
  };
```

The image created in Figure 4.5 is now static, as the parameters are not changing in the animation loop (the draw function). Also, the parameter values are not changed by any input data stream at this moment. Let's see how the animation changes when the parameter "radius" is changed in the visual code. For every new frame, the radius is increased by 1 and the circle appears larger in the output canvas:

```
// Variables
const cellSize = 50; // Size of each cell
var circleSize = cellSize // initial size of each circle

const cols = Math.ceil(visualWidth / cellSize);
const rows = Math.ceil(visualHeight / cellSize);

// DRAW FUNCTION
p.draw = () => {
  p.background(0);
  p.translate(-visualWidth / 2, -visualHeight / 2);

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const x = j * cellSize + cellSize / 2; // X-coordinate of the circle's center
      const y = i * cellSize + cellSize / 2; // Y-coordinate of the circle's center

      // Draw a circle at the calculated position
      p.circle(x, y, circleSize);
    }
```

```
  }
  circleSize += 1
};
```

There we are: the image now has changing circle radius so that it is a dynamically changing image. The animation automatically starts in the output canvas from the moment the visualisation code is valid.

4.5.2 Input selection

The previous section explained how an image could be created and tested in the web browser tool, and how this image could be animated through time. To continue, the animation's movements can be triggered by external factors. For example, can the radius be changed on a user configurable event? Let's investigate the specific case where the radius increases when the user presses the "arrow up" key, and decreases when the user presses the "arrow down" key. To implement this, the developer can configure it's first input data streams, which are in this case two streams of key-press events: one for Arrow up and one for Arrow down. This can be translated into the following nodes:

- Observable1: Key press Arrow Up

```
return rxjs.fromEvent(document, 'keydown')
    .pipe(
  rxjs.filter(event => event.code === 'ArrowUp')
 )
```

- Observable2: Key press Arrow Down

```
return rxjs.fromEvent(document, 'keydown')
    .pipe(
  rxjs.filter(event => event.code === 'ArrowUp')
 )
```

- Observer 1: callback function Arrow Up: circle radius + 1

```
return obs["keyUp"].subscribe(
  () => { params["radius"] += 10; }
)
```

- Observer 2: callback function Arrow Down: circle radius - 1

```
return obs["keyDown"].subscribe(
  () => { params["radius"] -= 10; }
)
```

- Parameter: radius

After creating these five nodes and pressing the RUN button, the following diagram is created:

When now pressing the Arrow up and Arrow down keys, the upper path and lower path of the graph respectively are highlighted and the radius is increasing and decreasing with 10.

If we want the animation to react on these user interactions, we need to use the value of params["radius"] as circle size inside of the visualisation code:

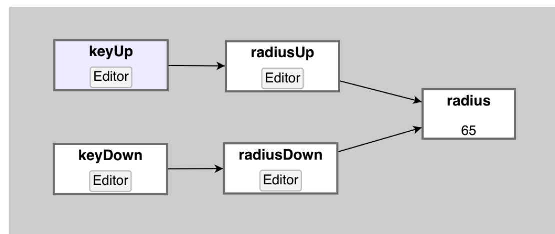

Figure 4.6: Diagram radius changes through key presses

```
// Variables
const cellSize = 50; // Size of each cell

const cols = Math.ceil(visualWidth / cellSize);
const rows = Math.ceil(visualHeight / cellSize);

// DRAW FUNCTION
p.draw = () => {
  p.background(0);
  p.translate(-visualWidth / 2, -visualHeight / 2);

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const x = j * cellSize + cellSize / 2; // X-coordinate of the circle's center
      const y = i * cellSize + cellSize / 2; // Y-coordinate of the circle's center

      // Draw a circle at the calculated position
      p.circle(x, y, params["radius"]);
    }
   }
  };
```

Here we go: we now have a visual that is reacting to user input in real time. The following screenshots show the animation for different radius parameter values:

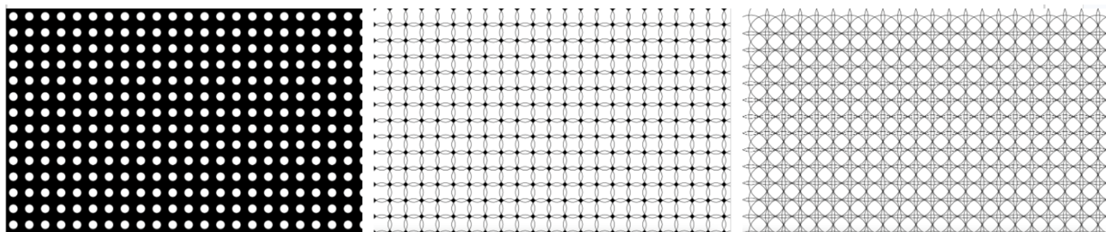

Figure 4.7: Animation of circles in grid

4.6 Challenges during development

In this section, the challenges faced during the development of the visualisation web tool are briefly discussed.

4.6.1 Layout

The first challenge was coming up with a nice and user friendly layout that enhances the creative process of the developer. The goal was to visually represent the connections between the input configuration, the visualisation code and the visual canvas. The interaction between these three components should be visible at all time, and switching from the one to the other component should be easy. For that reason, the layout of expandable components was chosen. This makes it possible to see the visual changing (in the output component) and see the parameter values evolving (in the input component) while coding in the visual code playground. At the same time, a detailed view on input nodes, code editors and visuals in full screen mode are supported. Further, a visual representation of external triggers influencing parameter values was desired. The implementation of the GOJS diagram and implementing the interactivity of the diagram nodes to different triggers (errors in the code, observables emitting new values, a node's editor being opened, ...) was challenging. In the end we succeeded to make the diagram interactive and transparent.

4.6.2 Performance and memory leaks

The creation of observables and observers happens when the developer clicks the RUN button. In RxJs, every observable and observer keeps on running as long as it is not shut down explicitly. At first, the tool didn't include the functionality of unsubscribing observers and completing observables when a new RUN was started, which lead to memory leaks and performance issues. Some refactoring and restructuring was needed to prevent these memory leaks and shut down correctly every observable and observer at the right time.

4.6.3 Communication between components

The communication and reuse of variable values between different components is implemented through dictionary objects. Here the challenge was to not lose user friendliness of the tool by requiring accessing parameters, observables and global variables by dictionaries, as this possibly makes the code less readable or longer. To address this, we assigned the dictionaries short names to minimize any inconvenience in the user's coding experience and quality.

4.6.4 Error messages of editors

Another challenge involved dealing with error messages and issues while running the code in the editors of observables, observers, and the visual playground. Ensuring that all errors were caught and properly presented below the corresponding editor proved to be quite demanding. Often uncaught errors still resulted in error messages in the browser console, without providing any indication of their origin to the developer. Moreover, crafting error messages that were understandable to the developer, given their execution amongst other back end application code, proved to be a challenging task. This is still an issue and can be improved in the future, to make debugging in the tool easier and more user friendly.

Chapter 5

Demos

Now that the web browser visualisation tool is extensively explained in all it's components and functionalities, it's time to create some visualisations interacting with music input streams. In this thesis, two demo's are created and explained. The first demo is a grid of cubes reacting on MIDI signals, coming from a MIDI piano connected to the user's computer. The second one is a 3D visual of turning torus objects, that reacts on an audio signal coming from the user's computer microphone.

5.1 Demo 1: cube grid piano

5.1.1 visualisation code

Let's first start with the code creating the visual, without taking into account the interactivity with the midi keyboard. Similar to the circle grid example, this visual shows a grid of p5 cubes. Each cube is turning around with a certain angle. To add some movements and colour changes around the grid, the 2D Perlin noise ([36]) function is used to add some randomness to the angle, so that every cube has a slightly different angle than it's neighbour cubes. Further, we add two p5 lighters to provide some colour in the cube grid. In this demo, we choose for a white point light and a blue directional light. The visual code looks like this:

```
const cellSize = 30; // Size of each cell
var cubeSize = cellSize - 10
const cols = Math.ceil(visualWidth / cellSize);
const rows = Math.ceil(visualHeight / cellSize);
var angle = 0

p.draw = () => {
  p.strokeWeight(0);
  p.background(0);
  p.translate(-visualWidth / 2, -visualHeight / 2);

  // create lights
  p.directionalLight(
    0,255,255, // color
```

```
    1, 1, 0  // direction
  );
  p.pointLight(
    255, 255, 255, // color
    40, -40,0  // position
  );

  // draw grid of cubes
  for (let i = 0; i < rows; i++) {
for (let j = 0; j < cols; j++) {
p.fill("white")

const x = j * cellSize + cellSize / 2; // X-coordinate of the cube's center
const y = i * cellSize + cellSize / 2; // Y-coordinate of the cube's center
const z = cellSize + cellSize / 2; // Z-coordinate of the cube's center

// Draw a cube at the calculated position
p.push(); // Save the current drawing state
p.translate(x, y, z); // Move to the cube's position
p.rotateX(angle + p.noise(i/5, j/5)); // Apply rotation around the X-axis
p.rotateY(angle + p.noise(i/5, j/5)); // Apply rotation around the Y-axis

p.box(cubeSize); // Draw the cube
p.pop(); // Restore the previous drawing state
    }
  }

  angle += 0.01

  };
```

The cubes are each turning around but with a slightly different angle, which results, in a combination with the lighting, in a moving colour gradient around the grid. This is seen in the following screenshots:

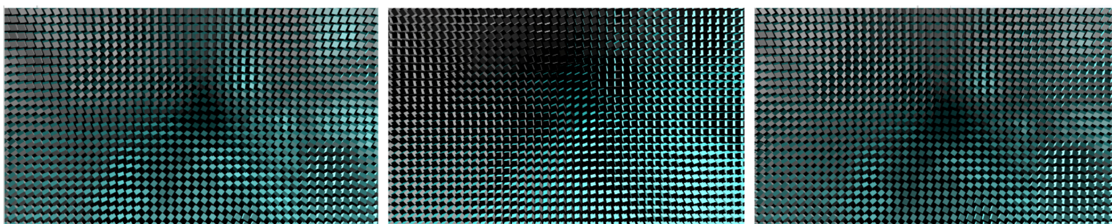

Figure 5.1: Cube grid piano: animation

5.1.2 Input selection

Now this animation will be spiced up by adding an input data stream coming from a MIDI keyboard. The keys pressed on the MIDI keyboard should be captured by an observable and

processed to parameter values influencing the visual. This is implemented using the Web MIDI API, which provides capabilities to work with MIDI devices directly from a web browser. It allows web applications to connect to MIDI controllers and send and receive MIDI messages. It results in the following observable code:

```
const midiInputsObservable = navigator.requestMIDIAccess().then(midiAccess => {
    const midiInputs = midiAccess.inputs
    const inputObservables = Array.from(midiInputs.values()).map(input =>
        rxjs.fromEvent(input, 'midimessage').pipe(
            rxjs.map(event => event.data)
        )
    )
    observableMerged = rxjs.merge(...inputObservables);
    return observableMerged
})
return midiInputsObservable
```

This observable listens to all MIDI devices connected to the computer and merges them into one observable with the RxJs merge operator. As explained in the background section on audio and midi signals, a MIDI signal is a tuple of three numbers: 1) action code (press or release), 2) note number and 3) velocity. Each tuple coming from the Web MIDI API is emitted by this MIDI observable.

To actually process this MIDI signal and transform it into a parameter change, an observer needs to be subscribed to this observable. This observer takes the seconds item in the midi signal array, extracts the minimal note key of the keyboard from it and assigns this value to a parameter pressedNote:

```
    minNoteKey = 48
    subscription = OBS["midiSignal"].subscribe(
        midiSignal => { PAR["pressedNote"] = midiSignal[1] - minNoteKey; }
)
```

The parameter pressedNote can have now values from 0 to the maximum note key of the keyboard. If we want to make the pressed note visible in the cube grid, the following condition can be added to the visualisation: code:

```
if (PAR["pressedNote"] === j)  {
    p5.fill("red")
} else { p5.fill("white") }
```

If we now play the piano, we see each note played appearing in a red column of cubes in the cube grid visual, as if the cube grid is a digital piano. This is just a simple interaction to start with, but the developer is free to come up with more complex or artistic interactions he/she can think of.

5.2 Demo 2: turning tori

5.2.1 visualisation code

As in the previous demo, we first create the visual without interaction of external data streams. This visual exists of multiple 3D tori with different size, turning around in a black screen.

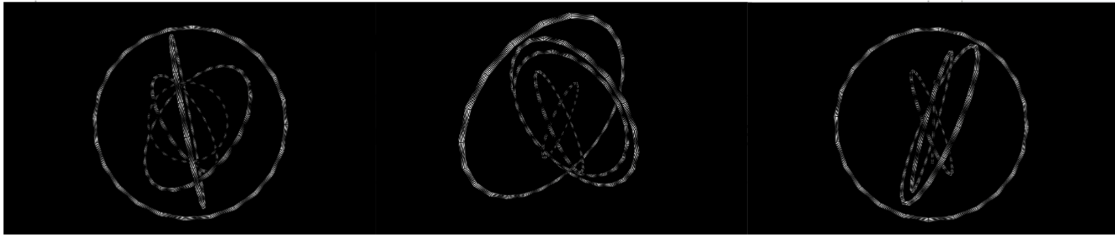

Figure 5.2: Turning tori: animation

We first create a class that describes a torus: an object with a size and a thickness, and three direction indicators, one for each direction in the 3D field. A torus object also has a draw function, that draws the torus on the canvas.

```
class Torus {
  constructor(size, thickness, colour, directionx, directiony, directionz) {
  this.size = size;
  this.thickness = thickness,
  this.colour = colour;
  this.angleX = 0;
  this.angleY = 0;
  this.angleZ = 0;
  this.xRotation = 0.01;
  this.yRotation = 0.01;
  this.zRotation = 0.01;
  this.directionX = directionx
  this.directionY = directiony
  this.directionZ = directionz
  }

  drawTorus() {
p.push(); // Save the current transformation state
p.fill(this.colour);

   p.rotateX(this.angleX);
p.rotateY(this.angleY);
p.rotateZ(this.angleZ);

   p.torus(this.size, this.thickness);
p.pop(); // Restore the transformation state
  }

}
```

Next, we create five tori, each with different properties like size and turning directions, and save them in an array.

```
var tori = []
tori.push(new Torus(visualHeight/2.5, 6, "white", 1, -1, -1));
```

```
tori.push(new Torus(visualHeight/3, 5.5, "white", -1, -1, -1));
tori.push(new Torus(visualHeight/3.5, 5,"white", 1, -1, -1));
tori.push(new Torus(visualHeight/4.5, 4.5, "white", 1, 1, 1));
tori.push(new Torus(visualHeight/6, 4, "white", -1, -1, 1));
```

Finally we write the draw function, that colours the background in black and loops over the torus array to draw each torus one by one in the screen. At the end of the draw loop, the three angles of each torus are updated with their respective rotation.

```
p.draw = () => {
p.background("black")
tori.forEach((torus) => {
torus.drawTorus();
torus.angleX += torus.xRotation * torus.directionX
torus.angleY += torus.yRotation * torus.directionY
torus.angleZ += torus.zRotation * torus.directionZ
})
```

5.2.2 Input selection

We create one observable that listens to the user's computer microphone via browser's Web Audio API. It returns a frequency array every 50 milliseconds.

```
const audioCtx = new (window.AudioContext || window.webkitAudioContext)();

const analyser = audioCtx.createAnalyser();
const freqArray = new Float32Array(analyser.frequencyBinCount);

const audioObservable = navigator.mediaDevices.getUserMedia({ audio: true })
  .then(stream => {
    const source = audioCtx.createMediaStreamSource(stream);
    source.connect(analyser);

    // Return the audio observable
    return rxjs.interval(50).pipe(
      rxjs.map(() => {
        analyser.getFloatFrequencyData(freqArray);
        return Array.from(freqArray);
      })
    );
  });

observable = audioObservable;
```

This frequency data can be transformed by an observer in many ways. In this demo, we chose for a simple transformation by dividing the frequency data into five chunks, and calculating the maximum (positive) frequency for each chunk. The result value is an array of five elements representing each the 'loudness' of each of the chunks.

```
subscription = obs["audioSignal"].subscribe(freqArray => {
  const normalizedArray = Array.from(freqArray).map(value => (value + 140) / 140);
```

```
const chunkSize = Math.floor(normalizedArray.length / 5);
const maxValues = [];

for (let i = 0; i < 5; i++) {
  const startIndex = i * chunkSize;
  const endIndex = startIndex + chunkSize;
  const chunk = Array.from(normalizedArray.slice(startIndex, endIndex));
  const max = Math.max(...chunk);
maxValues.push(max);
params["loudness"] = maxValues.map(value => parseFloat(value.toFixed(2)))
}
})
```

The last line of the observer code sets the array of highest frequencies for each chunk to a parameter called ”Loudness”. This parameter controls how fast the tori in the visual rotate. The bigger torus represents the loudness of the low-pitched sounds, while the smaller one represents the loudness of the high-pitched sounds. This creates a interactive visualisation in real-time of the sounds picked up by the user’s computer.

Chapter 6

Evaluation and testing

The web application and the strengths and weaknesses of p5.js in combination with reactive programming are evaluated extensively by combining insights coming from 1) previous research insights explored in the related work section, 2) insights during the process of the creation of the demo visualisations, and 3) feedback received from a group of five test persons who explored and tried out the web application in all it's functionalities.

During the testing phase, the application's functional documentation was provided within the application itself, and the application was hosted on a GitHub Page. This allowed all testers to conveniently access the tool using their preferred web browser. At the commencement of the testing phase, all test participants received an explanation of the web application and its functionalities. The evaluation occurred through both interviews and surveys featuring open-ended questions.
The results are detailed in the sections below.

6.1 Application concept in general

Many test users highlighted the application's intuitive layout and the straightforward conceptual logic of its various building blocks. The user-friendly nature of the application enabled the swift and effortless creation of simple visuals, incorporating basic interactions such as key presses or time intervals. This lowered the barrier for first-time users to engage with interactive visual creation. One tester indicated he enjoyed to explore the possibilities in iterations: by first creating simple and then more complex and interactive visuals in a playful manner. Another test user came up with the idea to create a game with it, showing the flexibility and versatile nature of the application.

6.2 Creating visuals in Javascript p5

Creating visuals with p5.js in the browser offers an accessible and user-friendly environment for creative coding. The p5.js library provides a simple and intuitive syntax, making it easy for developers that are new in the field of visualisations, to start creating visuals quickly. With p5.js, developers can generate animated visuals and create both 2D and 3D graphics within an HTML canvas element, providing a versatile canvas for creative expression. All test users, both with technical and non technical background, agreed on the user friendliness and accessibility

of the p5 framework. One test user suggested to flatten the learning curve further by providing coding support (examples, illustrations) and documentation links of p5.js in the web application.

One of the significant advantages of using JavaScript for music visualisation is the testability and real-time interaction it offers. Being browser-based, p5.js allows developers to see immediate visual feedback, making it highly testable during the development process. This real-time interaction capability enables quick iterations and adjustments, fostering a more efficient creative workflow. One test user indicated the focus on fast realisation, testing and trial and error in the tool as a strength in the creative process.

One test user with some experience in creating visuals with Processing (related to p5) indicated that the choice for a browser tool and javascript p5 is logic and accessible. JavaScript runs on almost all modern web browsers, ensuring broad accessibility and compatibility for users across different platforms and devices. The ubiquity of web browsers on desktop and mobile devices positions p5.js as a "write once, run anywhere" solution for developing musical interfaces. This portability and distribution ease allow creators to reach a wider audience without requiring users to install specific software. This also became clear during testing with different test people: the setup time of the tool was very minimal, and also sharing created interactive visuals could easily be shared and run on different devices.

Additionally, JavaScript has a vast and active community, providing developers with extensive resources for learning, problem-solving, and sharing creative projects. This active community contributes to the continual improvement and evolution of JavaScript and its associated libraries. This was also noticed by two test persons during the development of their demo visualisation.

However, using JavaScript for music visualisation also comes with a limitation regarding performance, especially when dealing with complex and computationally intensive visualisations. During the development of the two demo's, that processed rather complex input data streams and animated visualisations, the performance boundaries were reached. In the second demo processing audio signals, the frequency of emitting audio data by the observable, had an influence on the visualisation movements, leading to a less smooth and efficient visualisation experience. In the first demo processing midi signals, the performance boundaries were reached when increasing the number of cubes in the visual, resulting in a more slowly moving visual. Next to the performance limitations of p5, also the Web Audio API showed it's limitations: one test person mentioned that the audio output lagged slightly behind the input by a fraction of a second. This "almost but not quite in sync" situation could pose a challenge for music.

In conclusion, using p5.js for interactive music visualisation provides an accessible and versatile platform for creating music visualisations in the browser. Its ease of start-up, real-time interaction and testability, intuitive syntax and large user community make it an attractive choice for developers that are new in the field of visualisation. However, it's crucial to be mindful of potential performance limitations when building more complex and reactive visualisations.

6.3 p5.js with reactive programming

The integration of reactive programming using RxJS in combination with the p5.js library for creating interactive visualisations in the browser has been a significant and enriching aspect of this master thesis. The ability to connect observables to the p5.js code and use them to adapt visual parameters based on emitted values has provided a new level of interactivity and dynamic responsiveness to the visuals. This integration allowed for real-time synchronisation between the music data flows and the visual output, providing engaging and interactive visual experiences.

By incorporating reactive programming principles, the web application enabled developers to easily configure and combine multiple input streams, such as MIDI keyboard presses, microphone inputs, or custom user interactions. This flexibility expanded the creative possibilities, allowing developers to experiment with diverse input sources and visualise the effects in real time. One test person highlighted the flexibility and modularity of the application, which makes it possible to think of interesting integrations like online microservices, sensors, or AI interactions. Furthermore, the combination of p5.js's graphics capabilities and RxJS's data stream manipulation provided a seamless blend of creative coding and reactive programming paradigms. This synergy offered a powerful and efficient approach to building interactive visuals, allowing developers to focus more on the creative aspects of visual creation rather than dealing with low-level technicalities.

The visual representation of the observables, observers and parameters in a linked diagram using GOJS added an intuitive and visual aspect to the reactive programming process. This integration with visual programming concepts enhances the overall user experience, making it easier for developers to understand the connections between different elements and understand the behaviour of the visuals in response to various inputs.

In evaluating the use of reactive programming with p5.js, it became clear that this approach significantly reduced the coding time and intuitiveness required to achieve interactive visualisations. The declarative nature of reactive programming with RxJs simplified the handling of complex data flows, making it easier to manage the codebase and apply changes or improvements. This reduction in complexity also facilitated the scalability of digital artworks, enabling the addition of new observables and interactive parameters with ease. However, as stated above, performance limitations of running p5.js in the browser need to be taken into account.

It is essential to note that while reactive programming provided substantial advantages in terms of interactivity and data flow management, it also introduced a learning curve for developers who were not familiar with the reactive programming paradigm. One test user noted that, thanks to this tool, they grasped the fundamentals of reactive programming (observers and subscribers) and p5 visualizations in a straightforward manner. However, some developers might initially find it challenging to understand and leverage the full potential of RxJS's capabilities effectively. Therefore, the integration of reactive programming in this web application should be accompanied by clear documentation and guidance to help developers familiarise themselves with the concepts. The need for clear examples and documentations links was also emphasised by multiple test users.

Additionally, certain aspects of the application, such as suboptimal error messaging in each JavaScript editor or the use of dictionaries for storing and accessing parameters and observables / observers, could be further improved to enhance user experience and reduce the learning curve. Also, a test user mentioned that understanding how the different components are working to-

gether in the background, can help to solve and understand error messages and problems faster. By addressing these areas of improvement and providing user-friendly interfaces and documentation, the application can become more accessible to developers of varying skill levels. As the development process is iterative, user feedback and testing can guide the implementation of these enhancements, ultimately leading to a more intuitive and efficient web application for creating interactive music visualisations.

6.4 Conclusion of test phase

In conclusion, the integration of reactive programming with p5.js in the web application has been a successful effort that enhanced the interactivity and creative possibilities. The combination of these two technologies proved to be a powerful and efficient approach for building interactive visuals that adapt in real time based on music data flows. The visual representation of data flows using GOjs further contributed to the user-friendliness and understanding of the visual's interactivity. However, to fully utilise the potential of reactive programming, providing appropriate learning resources and documentation will be crucial for developers aiming to leverage this approach effectively. Furthermore, performance limitations need to be taken into account when building large installations. Overall, this integration has demonstrated the possibilities and advantages of combining reactive programming with the p5.js library for creating interactive and dynamic music visualisations in the browser.

Chapter 7

Conclusion

This thesis delves into the realm of interactive visual creation. Numerous technologies, languages, and methods exist for crafting interactive visuals, catering to diverse skill sets and backgrounds. The abundance of options can be overwhelming, and this thesis aims to identify technologies that are approachable, providing a delightful introduction to music visualisation for new users in creative coding.

In this research, a web application is created that combines the JavaScript's p5.js framework with the principles of reactive programming. The web application provides functionality to draw p5 images and animations in a HTML canvas easily, and with real time output or error messaging. Furthermore, interactivity of the visual can be added by connecting parameters in the visualisation code to user configurable input streams with the help of the RxJs reactive programming framework of Javascript. Inspired by the principles of visual programming languages, a visual graph representation shows the structure and impact of this input stream configuration and how it changes values of visual parameters.

During the development and in the test phase of the application, the benefits and challenges of the web application and of combining p5.js and reactive programming for creating interactive visuals were analysed. A first key takeaway is the testability and real-time interaction the browser tool offers, enabling quick iterations and adjustments and fostering a more efficient creative workflow. The web application highlighted the synergy between p5.js and reactive programming as a powerful combination for crafting interactive music visualisations. The application's user-friendly layout and intuitive building blocks lowered the entry barriers, enabling fast creation of simple to complex visuals with basic interactions. The integration of p5.js with reactive programming using RxJS showcased the power of real-time synchronisation between music or user interaction data streams and visual outputs, offering dynamic and engaging visual experiences. In addition to the benefits of this integration, performance considerations were noted for complex visuals, and the learning curve associated with reactive programming was acknowledged. Feedback from test users further underscored the importance of clear documentation and supportive resources to maximise the potential of this approach.

In summation, this research explores the potential of merging p5.js with reactive programming to create interactive music visualisations, improving interactivity, responsiveness, and creative expression. It provides a foundation for developers new in creative coding to explore the fascinating realm of developing interactive music visualisations.

Bibliography

[1] Climate spiral: Global temperature over time, n.d. URL https://climate.nasa.gov/climate_resources/300/video-climate-spiral-1880-2022/.

[2] MiMi. What is data art?, 2023. URL https://agoradigital.art/what-is-data-art/.

[3] E. Sandberg. Creative coding on the web in p5.js: A library where javascript meets processing, 2019.

[4] P. P. Ray. A survey on visual programming languages in internet of things. *Scientific Programming*, pages 1–6, 2017. doi: 10.1155/2017/1231430.

[5] Refik Anadol. Bosphorus : Data sculpture. URL https://refikanadolstudio.com/projects/bosphorus-data-sculpture/.

[6] M. Akten. Simple harmonic motion #8, n.d. URL https://www.memo.tv/works/simple-harmonic-motion/.

[7] This is Colossal. Scale collective flux, 2021. URL https://www.thisiscolossal.com/2021/04/scale-collective-flux/.

[8] A. Pošćić, G. Kreković, and A. Butković. Desirable aspects of visual programming languages for different applications in music creation, 2015. URL https://www.researchgate.net/publication/331783205_Desirable_Aspects_of_Visual_Programming_Languages_for_Different_Applications_in_Music_Creation.

[9] A. Pošćić and G. Kreković. Ecosystems of visual programming languages for music creation: A quantitative study. *Journal of the Audio Engineering Society*, 66(6):486–494, 2018. doi: 10.17743/jaes.2018.0028.

[10] Some techniques on digital signal processing in the visual programming environment max, 2020. URL https://ieeexplore.ieee.org/abstract/document/9131297/figures#figures.

[11] M. C. Negrão. Nndef: livecoding digital musical instruments in supercollider using functional reactive programming, 2018. URL https://doi.org/10.1145/3242903.3242905.

[12] C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin. Designing musical instruments for the browser. *Computer Music Journal*, 39(1):27–40, 2015. doi: 10.1162/comj_a_00283.

[13] Towards Data Science. Understanding audio data: Fourier transform, fft, spectrogram, and speech recognition, n.d. URL https://towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d2

[14] Audio Interfacing. Midi tracks vs. audio tracks, n.d. URL https://audiointerfacing.com/midi-tracks-vs-audio-tracks/.

[15] G. Forsberg. An audio-to-midi application in java, 2009.

[16] ResearchGate. An example of midi message, n.d. URL https://www.researchgate.net/figure/An-example-of-MIDI-message_fig1_343709022.

[17] GeeksforGeeks. Difference between digital audio and midi, n.d. URL https://www.geeksforgeeks.org/difference-between-digital-audio-and-midi/.

[18] T. Murphy. A livecoding semantics for functional reactive programming, 2016. URL https://doi.org/10.1145/2975980.2975986.

[19] University of Auckland. What is visual programming?, 2005. URL https://www.cs.auckland.ac.nz/courses/compsci732s1c/archive/2005/lectures/WhatIsVP.pdf.

[20] Jean-François Charles. A tutorial on spectral sound processing using max/msp and jitter. *Computer Music Journal*, 32(3):87–102, 2008.

[21] Bryan WC Chung. *Multimedia Programming with Pure Data.* CRC Press, 2013. ISBN 978-1466565996.

[22] Xiangnuo Li and Jialu He. Exploring the emotional design of digital art under the multimodal interaction form. In *Lecture Notes in Computer Science*, volume 14030, July 09 2023. URL https://link.springer.com/chapter/10.1007/978-3-031-35699-5_40.

[23] Luc Nijs, Pieter Coussement, Chris Muller, Micheline Lesaffre, and Marc Leman. The music paint machine - a multimodal interactive platform to stimulate musical creativity in instrumental practice. In *Conference Paper*, Institute for Psychoacoustics and Electronic Music, Ghent University, Blandijnberg 2, 9000 Ghent, Belgium, January 2010. URL https://www.researchgate.net/publication/221130789.

[24] Emily Zimmerman and Amir Lahav. The multisensory brain and its ability to learn music. *Annals of the New York Academy of Sciences*, 2012. doi: 10.1111/j.1749-6632.2012.06455.x. URL https://nyaspubs.onlinelibrary.wiley.com/doi/abs/10.1111/j.1749-6632.2012.06455.x.

[25] Melissa Bremmer, Carolien Hermans, and Vincent Lamers. The charmed dyad: Multimodal music lessons for pupils with severe or multiple disabilities. *International Journal of Music Education*, 43(2), 2022. doi: 10.1177/1321103X20974802. URL https://journals.sagepub.com/doi/abs/10.1177/1321103X20974802.

[26] Martin Klasen, Yu-Han Chen, and Klaus Mathiak. Multisensory emotions: perception, combination and underlying neural processes. *Reviews in the Neurosciences*, August 2012. doi: 10.1515/revneuro-2012-0040. URL https://www.degruyter.com/document/doi/10.1515/revneuro-2012-0040/html.

[27] Anna Fekete, Rosa M. Maidhof, Eva Specker, Urs M. Nater, and Helmut Leder. Does art reduce pain and stress? a registered report protocol of investigating autonomic and endocrine markers of music, visual art, and multimodal aesthetic experience. *PLoS ONE*, 17(4):e0266545, April 2022. doi: 10.1371/journal.pone.0266545. URL https://doi.org/10.1371/journal.pone.0266545.

[28] A. M. Stark. Sound analyser: A plug-in for real-time audio analysis in live performances and installations, 2014. URL https://www.semanticscholar.org/paper/Sound-Analyser%3A-A-Plug-In-for-Real-Time-Audio-in-Stark/ee6288d289d200b2e221634c6344083e04bb7554.

[29] M. Graf. An audio-driven system for real-time music visualisation. 2021. URL https://arxiv.org/abs/2106.10134.

[30] J. Bresson, C. Agon, and G. Assayag. Openmusic: Visual programming environment for music composition, analysis and research. In *Proceedings of the 19th ACM International Conference on Multimedia*, pages 743–746. Association for Computing Machinery, 2011. doi: 10.1145/2072298.2072434.

[31] F. Collopy, R. M. Fuhrer, and J. D. Visual music in a visual programming language, 1999. URL https://doi.org/10.1109/vl.1999.795882.

[32] R. Taylor. Real-time music visualization using responsive imagery, 2014. URL https://www.academia.edu/7767058/REAL_TIME_MUSIC_VISUALIZATION_USING_RESPONSIVE_IMAGERY.

[33] C. Pramerdorfer. An introduction to processing and music visualization, 2011. URL https://www.semanticscholar.org/paper/An-Introduction-to-Processing-and-Music-Pramerdorfer/8794c8548fb56359ee44ac76e854dac464f61350.

[34] Interactive music visualization for music player using processing, 2016. URL https://ieeexplore.ieee.org/document/7863205.

[35] Research and implementation of real-time music visualization system based on python, 2020. URL https://doi.org/10.1117/12.2654065.

[36] p5.js contributors. p5.js reference: noise(), 2021. URL https://p5js.org/reference/#/p5/noise.