Highlighted portions generally need a rewrite

SonicSketch – composing with drawing metaphors in the modern web browser

Kevin Nolan

Music & Media Technologies
School of Engineering
&
School of Drama, Film and Music
Trinity College Dublin

Submitted as part fulfilment for the degree of M.Phil.

2017

Declaration

I hereby declare that this thesis has not been submitted as an exercise for a degree at this or any other University and that it is entirely my own work.

I agree that the Library may lend or copy this thesis upon request.

Signed: ____________________ Date: ____________________

Abstract

SonicSketch is a graphic synthesis music tool that runs in the modern web browser and allows users to draw sonic material on a canvas in a freeform, creative and intuitive manner.

It is aimed at the ideation phase of digital music creation and seeks to highlight the metaphor of sketching instead of the more common analog studio metaphors found in the digital audio workstation (DAW).

While it takes influence from a number of graphic synthesis systems SonicSketch directly builds on the ideas and concepts of SonicPainter, a separate application created by William Coleman. A critique of this system is given to identify the areas that SonicSketch improves upon.

A detailed account is given of the development process of SonicSketch, including the early prototype work, the architectural setup and the building of the application.

The resulting artifact is evaluated both from a self-assessment and a user perspective. The results of a usability test and general user feedback indicate that it is enjoyable, easy to use and learn.

Finally, a number of proposed improvements and features are given, to make the application a more flexible and useful tool for musicians.

Acknowledgments

Thanks to my family and friends for all the support throughout the past year, in particular this last stretch.

Thanks to the MMT staff and lecturers for their tireless dedication. In particular, thanks to Liam for guiding me through the sometimes stormy waters of completing this thesis project.

Finally thanks to all my classmates for making this a memorable and enjoyable year!

Contents

Abstract **iii**

Acknowledgments **iv**

1 Introduction **1**

1.1 Introduction 1
1.2 Motivation 2
1.3 Goals 2
1.4 Approach 3
1.5 Thesis structure 4

2 Background: Sonic Sketching **5**

2.1 Introduction 5
2.2 Dominant DAW metaphors 5
2.2.1 Multi-track tape recorder 5
2.2.2 Multi channel mixing desk 6
2.2.3 Outboard effects unit 7
2.2.4 The piano roll 8
2.3 A compositional example 8
2.3.1 Realization in a DAW 9
2.3.2 Realisation in code 11
2.4 Problems with flexible systems 12
2.5 Sketching as an alternative metaphor 13
2.5.1 Oramics 13
2.5.2 UPIC 14
2.5.3 A Golan Levin's AVES 16
2.5.4 William Coleman's sonicPainter 18
2.6 Conclusion 19

3 Methodology **20**
3.1 Introduction 20
3.2 Assessment of SonicPainter 20
3.2.1 Usability issues 21
3.2.2 Issues with audio visual connection 21
3.2.3 Invalid states 22
3.2.4 Linear representation of frequency 22
3.3 Technical approach 23
3.3.1 Audio in the modern Web Browser 23
3.3.2 Tone.js 23
3.3.3 Paper.js 24
3.3.4 React framework 24
3.3.5 Clojurescript 25
3.3.6 Reagent & Re-frame 27
3.4 Conclusion 28

4 Execution **29**
4.1 Introduction 29
4.2 Early prototype work 29
4.2.1 MelodyPainter 29
4.2.2 SonicShaper 30
4.2.3 Web version of William Coleman's SonicPainter 31
4.3 Setting up the architecture 32
4.3.1 Clojurescript and javascript npm modules 32
4.3.2 Paper.js and React (Paper.js bindings) 33
4.3.3 Tone.js and React 34
4.3.4 Reagent and React paper.js bindings 34
4.4 Core functionality - sketching notes 35
4.4.1 Adding notes 36
4.4.2 Editing notes 42
4.5 Secondary functionality 47
4.6 Conclusion 48

5 Evaluation **49**
5.1 Introduction 49
5.2 Returning to the composition 49

5.3 Assessment . . . 50
5.3.1 System architecture . . . 50
5.3.2 Core functionality - sketching notes . . . 50
5.3.3 Timbre selection . . . 51
5.3.4 Vibrato . . . 51
5.3.5 Other tools . . . 52
5.3.6 Performance issues . . . 52
5.4 User testing and feedback . . . 54
5.4.1 Usability questionnaire . . . 54
5.4.2 General feedback . . . 55
5.5 Conclusion . . . 56

6 Conclusion and further work **59**
6.1 Future work . . . 59

Bibliography **61**

Appendices **65**

A SUS questionnaire **66**

B Csound composition **67**

C Data disk index **69**

List of Figures

2.1 SSL SL9000J (72 channel) console at Cutting Room Recording Studio, NYC . . . 6
2.2 Skeuomorphic software FX . . . 7
2.3 DAW realisation of composition . . . 10
2.4 Daphne Oram's Oramics machine . . . 13
2.5 Iannis Xenakis - Mycenae Alpha score . . . 15
2.6 Iannis Xenakis showing UPIC to a younger audience . . . 16
2.7 Golan Levin's Aurora (part of AVES) . . . 17
2.8 SonicPainter by William Coleman . . . 18

4.1 MelodySketch interface . . . 30
4.2 SonicShaper interface . . . 31
4.3 SonicPainter in a web browser . . . 32
4.4 Adding notes in SonicSketch . . . 36
4.5 A note in SonicSketch . . . 41
4.6 SonicSketch tools panel . . . 43
4.7 Live coding the vibrato visualisation . . . 45

5.1 The Klangfarbenmelodie composition realised in SonicSketch . . . 50
5.2 SonicSketch sound palete . . . 51
5.3 Performance debugging in Google Chrome . . . 53
5.4 Heat graph displaying note start points . . . 55
5.5 Some exhibit participants managed to draw figurative artwork! . . . 56
5.6 Some users explored visual rhythms . . . 57
5.7 Vibrato added to a sketch . . . 57
5.8 Exploring a clustered approach to sonic composition . . . 58

Chapter 1

Introduction

1.1 Introduction

The traditional route for both recorded and performed musical expression has been through playing a musical instrument. Through this simple but powerful interface, ideas can be explored by the skilled practitioner in a free flowing, intuitive manner. The ubiquity of high powered computers has led to a fundamental change to this process, however. Electronic music can now be produced entirely on a computer, enabling those without the background or training to produce music to a professional standard. The affordances of infinite editability and tweak-ability allows the novice producer to organically build their skills by composing using a process of trial and error. The requirement to perform on a musical instrument to a high standard is no longer necessary. (Duignan, 2008 pg. 12) [Duignan, 2008, pg. 12]. Correct all references

The primary tool of this new breed of electronic musician, is the Digital Audio Workstation (DAW), a powerful software application for composing, arranging and editing musical events and audio material, mixing tracks and applying audio effects. To control this power, the electronic musician must master the vast array of knobs, sliders, buttons and complex hierarchies of settings. This complexity poses some difficult Human Computer Interaction (HCI) challenges both for software developers and end users (Duignan, Noble and Biddle, 2010).

A common solution to this is to provide the user with real world metaphors in the presentation of the interface. In this way, the highly abstract process that takes place in digital systems can be grounded in a metaphorical language understandable to the user. An extremely dominant metaphor is that of the analog mixer and the hardware multi-track tape machine which have, since the mid-1990s, become a mainstay construct in the interfaces of commercial DAWs (Bell et al. (2015)). The effectiveness of this metaphor is evident in the vast amount of music produced in this environment. There are some situations, however, where the metaphor can get in the way and lead to another new, separate layer of complexity that must, in turn, be managed by its users.

1.2 Motivation

This leads to the central motivation behind the work on SonicSketch: that of exploring alternative metaphors that support the early stage of music production and fulfills the role traditionally filled by an instrument. In exploring these alternative metaphors, an attempt is made to regain some of what is lost when working directly with the computer without compromising the distinct advantages it brings.

The intention is not to suggest that analog studio metaphors can be replaced, or even that there is any reason that they should be. Instead, the intention is to augment these rich metaphors with other metaphors more suited to certain stages of the creative process. The need for such efforts is in some ways acknowledged by mainstream DAW manufacturers. For instance Ableton Live, a highly popular DAW application, has recently opened its application programming interface (API) to allow other programs to create compatible files (Ableton, 2017). This enables producers to work in diverse, idiosyncratic and perhaps less featured environments more suited to the capturing of ideas and creative flow. The option is open to then move freely to the more powerful fully featured environments offered by the DAW.

1.3 Goals

The overarching and primary goal of this project is to create a software application called SonicSketch that is specifically targeted at the early ideation part of the music production process. Building on a strong theoretical and technical foundation it will forego densely populated analog studio inspired DAW interface patterns and will instead focus on the metaphor of sketching. Users will draw various graphic symbols and lines on screen to produce sounds of different pitch, amplitude, and timbre which can be played back and altered in realtime.

The final artifact should be:

Beginner friendly the user should not need any lengthy explanation or instructions on its use but should be able to dive in and start making sounds straight away.

User friendly basic standard usability features should be added to reduce friction and allow users to engage more freely with the application.

Expert friendly flexible enough to allow users to use it for longer periods of time without tiring of it.

These aspects will be assessed by using self-assessment as well as by testing with users. User testing will take the form of both interview and a standard usability test. To maximize availability and broaden the potential user base of testers, the application will be developed as a web application that will run in a modern browser without requiring the installation of any special software.

1.4 Approach

The project is based upon and builds on SonicPainter, an application built by William Coleman as a Master's thesis (Coleman, 2015). Similarly motivated by the overwhelming complexity of modern DAWs, the interface aims to be minimal and free of distraction. It presents a canvas like space that allows the user to draw lines of various length, shape, and orientation resulting in sounds that vary in timing, duration, frequency, and timbre.

Building on this existing work is advantageous in several ways. It provides a more concrete foundation than starting off with a blank slate. Features and approaches can be assessed so that certain features that work well can be incorporated and improved. Equally pitfalls in the original work can be avoided. A summary of intended improvements are as follows:

- Increased discoverability of functions by adding user interface elements that enhance the "sketch" metaphor.
- Increased accessibility by making it available online.
- Improvements to usability such as allowing users to sketch in a freehand way more easily.

- Technical improvements to avoid crashes and unexpected application behaviour.
- Improve correlation between the visuals and the audio.

1.5 Thesis structure

Chapter 2 begins with a discussion of the current dominant tools and practices in use in music production today with a strong focus on the ideation phase. Some alternative approaches are then discussed, focusing on systems that, to some degree, use sketching as a metaphor. Both legacy and more recent systems will be considered in this survey. A discussion and critique of SonicPainter will follow this. The technical approach being taken by SonicSketch will then be introduced followed by a detailed walkthrough of it's development. An evaluation of the success of the project will then be given both from the perspective of the creator and from the users that tested it. Finally, the broader implications of the work will be discussed in addition to some suggestions for future research and development.

Chapter 2

Background: Sonic Sketching

2.1 Introduction

This chapter begins with a discussion on the dominant metaphors present in the modern DAW and expands on the earlier introduction to the topic. A compositional example is given to illuminate the issues and limitations that these metaphors can impose on its users. This is followed up with a survey of legacy systems that take a graphical approach to their interface and can be conceptually framed with the metaphor of sketching.

2.2 Dominant DAW metaphors

As has been introduced, analog studio metaphors of tape machines and hardware mixing desks dominate the UI approach to DAW interface design. Other prevalent metaphors often found in these interfaces are that of the *outboard effects units*, and the *piano roll* (Levin, 2000; Bell, Hein and Ratcliffe, 2015). A description of these now follows.

2.2.1 Multi-track tape recorder

Multi track tape recording was introduced into the recording studios in the 1960s and is typified by the systems produced by Ampex and Studer. These allowed producers to record multiple tracks of audio information which could be edited

Provide links to brand and product pages

emphasis using italics or similar when introducing a term or jargon and using dubbing techniques mixed to taste. This unlocked significant creative possibilities and made albums like "Sgt Peppers Lonely Heart Club Band" possible. The underlying model of tracks typically manifests itself in DAWs as rectangular blocks stacked from top to bottom and running from left to right. Similar to editing tape, these can be spliced, cut, and pasted. Terms and techniques prevelant in DAWS like *bouncing*, *overdubbing* and *markers* (which were originally created using a physical pen), all have their roots in their analog precedents.

Use consistent hyphenation

2.2.2 Multi channel mixing desk

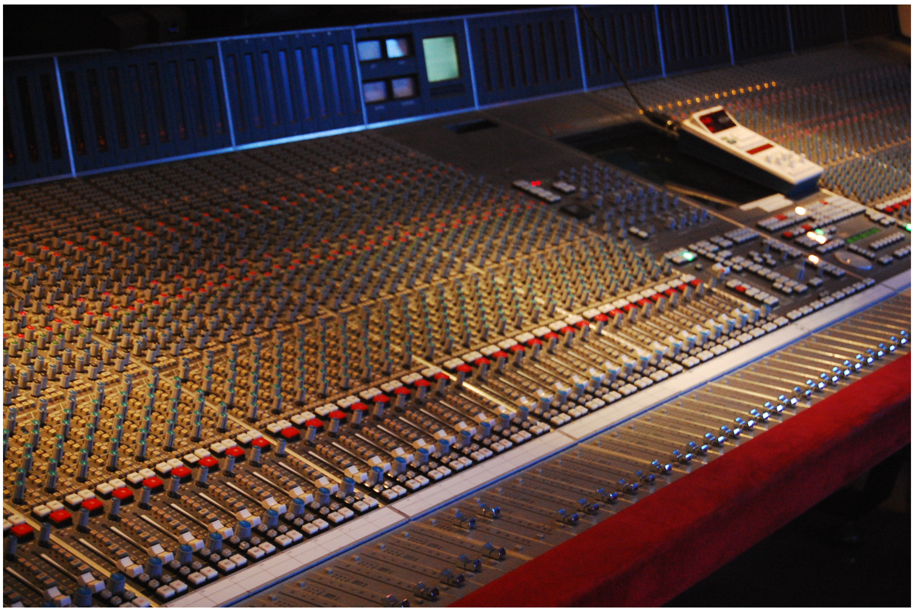


Figure 2.1: SSL SL9000J (72 channel) console at Cutting Room Recording Studio, NYC

The multi channel mixing desk metaphor is present in the large majority of DAWs and is normally represented in a similar fashion to the sliders (or faders) found in hardware mixing desks (see figure 2.1). The mixing desk enabled the producer to control the relative amplitude of a finite amount of channels in addition to performing tasks such as panning to balance the signal in a stereo field. The slim vertical sliders found on most systems were codified in the 1960's by Bill Putnam (Bell, Hein and Ratcliffe, 2015). This layout allowed the producer to manipulate multiple channels of audio simultaneously in a practice known variously as "riding the faders" and "playing the mixer". Despite the fact that the digital variants of these are largely controlled by a mouse that only affords the manipulation of a single fader at a time, they are still, largely speaking, presented in this fashion on screen.

2.2.3 Outboard effects unit

Figure 2.2: Skeuomorphic software FX

Outboard effects are hardware units used in studios to add audio effects to one or more channels on the mixing desk. The standard configuration of most studios allows for two different ways of applying these effects, by using insert effects and send effects. Insert effects are typically used when it only needs to affect a single channel, for instance, a chorus effect applied on an electric guitar. Send effects allow the producer to send a certain amount of the signal from a channel to specialised channels to perform processing on the signal in parallel with the original signal. It is typically used to apply effects on multiple channels of audio such as reverb. The introduction of Virtual Studio Technology (VST), by Steinberg, was responsible for bringing the outboard effects metaphor to a whole new height. This allowed third party developers to create virtual effects and instruments, and let producers expand their virtual studio beyond the built in effects. The visual interfaces increasingly paid homage to their hardware influences, emulating not only functionality but also the visual look (see figure 2.2). Bell et al. (2015) describe this as skeuomorphism, a design pattern where visual objects not only mimic real world objects in functionality but

link

also incorporate unneeded visual features. The purpose of this is not only decorative but also educational, and gives connotational cues as to how it should be used.

2.2.4 The piano roll

The piano roll is a primary metaphor found in almost all mainstream DAWS and is typically used to represent MIDI musical note information. MIDI, which stands for Musical Instrument Digital Interface is a standard protocol developed in the 1980s to allow control instructions to be sent between devices. It provided a standard language to for instance tell a synthesizer to play a particular note at a precise time and duration. These instructions could be collected into a MIDI file to, in effect, create a playable digital score. The piano roll is slightly distinct from the previously discussed examples in that it originates from a much earlier time period, the player pianos of the 1920s. The original piano rolls were operated by feeding a roll of paper with holes punched to indicate the precise timing that an attached piano should strike its notes. This provides an apt and suitable description for the MIDI musical data it normally represents. Similar to a player piano, no audible results are possible without an attached piano, and in the case of MIDI, an attached sound generating synthesizer device. (Levin, 2000; Bell, Hein and Ratcliffe, 2015)

2.3 A compositional example

Rather than discussing the issues that can arise from the metaphors in the abstract, let us consider a compositional idea and how we might achieve this in a DAW. The idea is broken into the following compositional “recipe”:

1. Two notes of the same timbre are played together about an octave apart for a duration of 2 seconds. portamento?
2. The first note glissandos to the frequency of the second note and vice versa.
3. The first note starts with a small amount of vibrato that quickly dissipates.
4. The second note starts with no vibrato but adds a small amount as the note nears completion.
5. When these two notes end, the same pattern is repeated except this time with different timbres and frequencies.

6. This is repeated 3 more times with different timbres and frequencies to complete this ten-second piece.

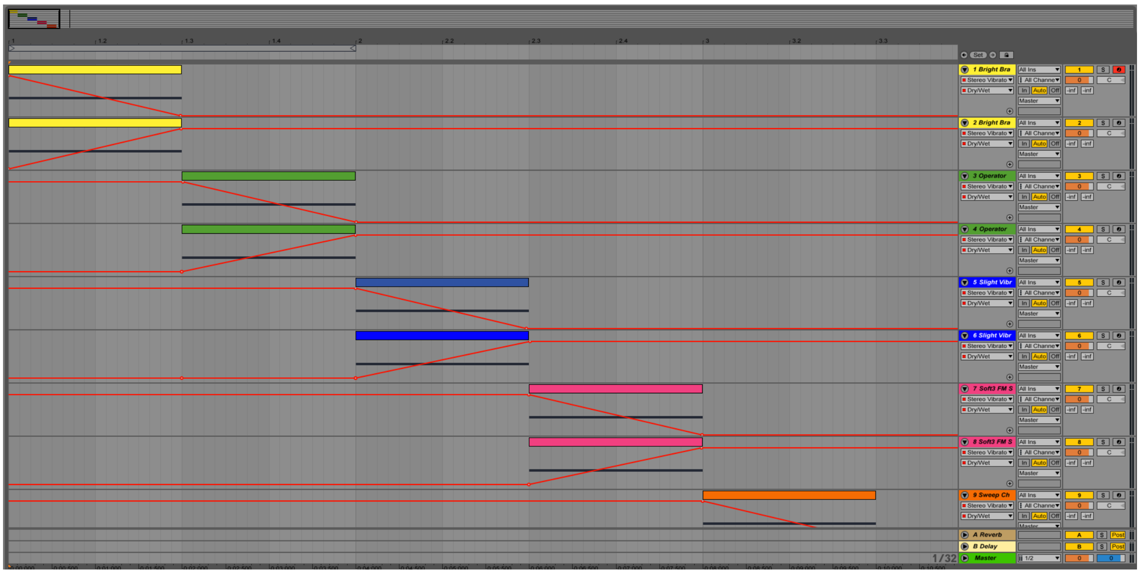
While this may seem like a contrived example this, in fact, constitutes a compositional technique called Klangfarbenmelodie (Cramer, 2002) that involves splitting a melodic line between instruments or timbres to create a timbre melody. The glissandos and altering of vibrato intensity add further complexity to better illustrate some of the weaknesses inherent in DAW metaphors.

2.3.1 Realization in a DAW

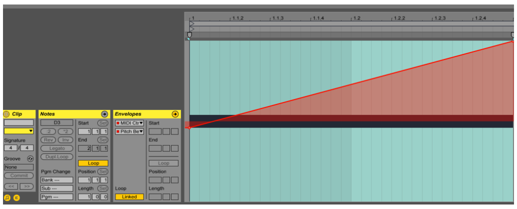
To achieve this in a DAW we have a few different options but a possible solution would be as follows:

1. Working with the multitrack tape metaphor we can create ten separate tracks to house two different versions of each timbre. A vibrato plugin effect should be added to each of these by using a send or an insert effect. Two different tracks are needed for each of the timbres due to the fact that the two notes are played at the same time and both have different frequency and effect trajectories. If on the other hand, they had the same effect modulations or were played at different times, no additional tracks would be needed.
2. Working with the piano roll metaphor, create a single note in each of these tracks setting each one to the desired fundamental frequency.
3. Now edit the pitch bend automation lane by clicking into the relevant dialog
4. Similarly, open the relevant dialog to edit the intensity of the vibrato effect
5. Repeat this for each of the notes in the composition.

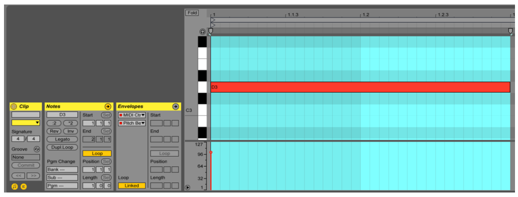
At this point, we may have achieved what we set out to do. However, we now may want to tweak each of these elements to taste and perhaps add more material. An explosion in track count and overall complexity is inevitable. This can lead to a ~~serious~~ slowdown in workflow, a loss of flow and cognitive overload. A common technique to combat this complexity overload is to bounce the tracks and then continue working on these simpler artifacts (Duignan, 2008). This, of course, negates a key advantage to working in a digital environment, the fine-grained ability to freely change, tweak and undo. Locating each note in separate tracks leads to an unnatural separation of what is, in fact, closely related compositional material. This requires



(a) The timeline showing the vibrato automation on each track.



(b) The pitch envelope. Requires multiple clicks to display.



(c) Each note is contained in its own timeline clip.

(d) The instrument and effect chain. The small red dot on the DRY/WET knob indicates automation.

Figure 2.3: DAW realisation of composition

awkward context switching and excessive navigation through the system to focus on different details.

There are of course other tools in the DAW that may achieve this task more easily. For instance, a sampler may allow us to use different timbres on the same track and may work better in this case. We now have the extra task of exporting each of these samples in preparation for our composition work. Some other options present in many DAWS include aggregate instruments, multi-timbral instruments, and perhaps some midi routing options. Another option is to use an alternative, more flexible, environment such as an audio programming language. Some brief consideration of this will now be given.

2.3.2 Realisation in code

link (correct all future omissions)

The piece could be realised in quite a straightforward manner in an audio programming language such as *Csound*. Central to *Csound* is the concept of the *unit generator* (or ugen), an abstraction to define both sound generators and processors. These can be patched together in a simple textual coding language to form instruments. A score is then specified, again in code, to define note onsets, durations in addition to other arbitrary parameters defined in the instruments. The required timbres and the vibrato effect could be made configurable on a per note basis by exposing these parameters. The Csound score could then trigger this instrument, with each note amounting to a single line of code, making the entire score a total of 10 lines. Full demonstration code is provided in the appendix. This compositional example will be revisited in a later chapter and discussed in the context of a further approach.

Table 2.1: CSound score represented as a table. Each row contains the data for each note.

Instrument id	Start time	Duration	Start freqeuncy	End frequency	Timbre	Start vibrato level	End vibrato level
1	0	2	440	880	3	.01	20
1	0	2	880	440	3	20	0.01
1	2	2	1320	660	6	0.01	20
1	2	2	660	1320	6	20	0.01
1	4	2	330	660	2	.01	20

Instrument id	Start time	Duration	Start freqeuncy	End frequency	Timbre	Start vibrato level	End vibrato level
1	4	2	660	330	2	20	.01
1	6	2	880	1720	1	.01	20
1	6	2	1720	880	1	20	.01
1	8	2	220	440	9	.01	20
1	8	2	440	220	9	20	.01

2.4 Problems with flexible systems

Depending on the experience of the reader, the realisation of the composition in code may or may not seem like a better approach than using a DAW. The reason for this is that this approach is not beginner friendly. An approach that is more forgiving in this regard is a pattern found in game design (Overholt, 2009). Games should be easy enough to get started without any special training or lengthy instructions but challenging enough to keep players engaged. Extremely open environments, such as that of an audio programming language, are not supportive of this initial onboarding of new users. This is not to say that it should only work for novices however (by for instance limiting pitches to simple scales). If a system is too closed it risks being more toy-like in nature and not supporting long term engagement (Wessel and Wright, 2017).

Perhaps a bigger criticism that could be made about open and complex systems, however, is that they can lead to an analytical rather than a creative way of thinking. In “Thinking Slow, Acting Fast”, Kahneman (2012) contrasts these two ways of thinking which he terms *System 1* and *System 2*. System 1 is instinctive, fast, emotional and is a mode of thinking that may not register consciously. System 2 is slow, logical, analytical and registers prominently in active consciousness. Routine tasks such as walking, opening doors etc only use system 1 thinking. These can be completed while exerting minimal cognitive effort (all the while calculating the complex motor sensory actions that must take place). Complex analytical tasks such as programming require system 2 thinking. Approaching creative tasks such as music making in this way where instinct and emotion are often crucial can slow down or stop the process. Perhaps it is best summed by John Cage: “Don’t try to create and analyse at the same time. They’re different processes” (Popova, 2012).

2.5 Sketching as an alternative metaphor

While audio programming languages offer a model that is closer to the underlying computational processes taking place than the more abstracted DAW interfaces. As we have discussed, though what is gained in flexibility can be lost in intuitiveness and ease of interaction. Rather than discarding these higher level metaphors, perhaps a better approach would be to explore alternate ones.

A rather promising but nonmainstream approach is that of sonic sketching. This has a long and illustrious historical precedent reaching back well before the, now more prevalent, studio metaphors. Graphical sound generation techniques have a long history starting with experiments beginning in the early 20th century (Roads, 1996 pg. 329). The technique of the optical soundtrack, however, brought these ideas to a new level of sophistication. The technique, which involved placing marks via photography or direct manipulation to specify audio properties, was explored by such luminaries as Oskar Fischinger, Norman McLaren and Daphne Oram. Oram’s particular take on the technique will now be discussed.

2.5.1 Oramics

Figure 2.4: Daphne Oram’s Oramics machine

A primary motivating factor behind Daphne Oram’s development of the Oramics machine was to bring more human-like qualities to the sounds generated by electronic means. The machine worked by playing back multiple lanes of film tape in unison, defining a monophonic series of notes as well as control signals to shape their timbre, pitch and amplitude. She details the thought process behind this in her journal style book, “An Individual Note” (Oram, 1972).

The aspects of the sound that she wishes to control are volume, duration, timbre, pitch, vibrato, and reverb. In order to do this, she describes a simple musical notation language based on the freehand drawing of lines combined with discrete symbols. The lines, which she describes as the analog control, are used to define volume envelopes. Interestingly, the default and preferred method for the parameters she wishes to control is the continuous line rather than discrete note symbols. For instance, she avoids the use of a static velocity per note and instead only specifies the use of a control envelope to change amplitude.

The discrete symbols, which she categorizes as digital control, are used to define individual pitches and are termed neumes. She highlights that notes should not remain static and, thusly, an analog control of each note is also specified. Similarly to amplitude and vibrato, timbre is also defined by the freehand drawing of lines and is something that with practice the “inner ear” can develop an intuition as the sonic results of different line shapes. It is Oram’s belief that the hand drawn nature of the lines make the results slightly inaccurate and to some extent unpredictable. Herein, however, lies the possibility of bringing more humanity to the cold and precise machines generating the electronic signal.

2.5.2 UPIC

The UPIC (“Unité polyagogique informatique du CEMAMU”) was a graphic sound synthesis system that was designed by Iannis Xenakis and arose from his graphic approach to composition. His earliest work, “Metastasis”, was conceived using a graphic approach to describe the trajectories and sound masses that inhabits the orchestral landscape of the piece. This approach has been attributed to his background in architecture, having worked in the studio of Le Corbusier. The UPIC was first conceived of in the seventies, with the realisation of the first version in 1975 and its first public showcase in 1977 (Roads, 1996 pg. 331). The work “Mycanae Alpha” (figure 2.5), composed in 1978 was the first work to use the system and was a “nine-minute 38-second composition of dense and intense textures, of phase-shifting

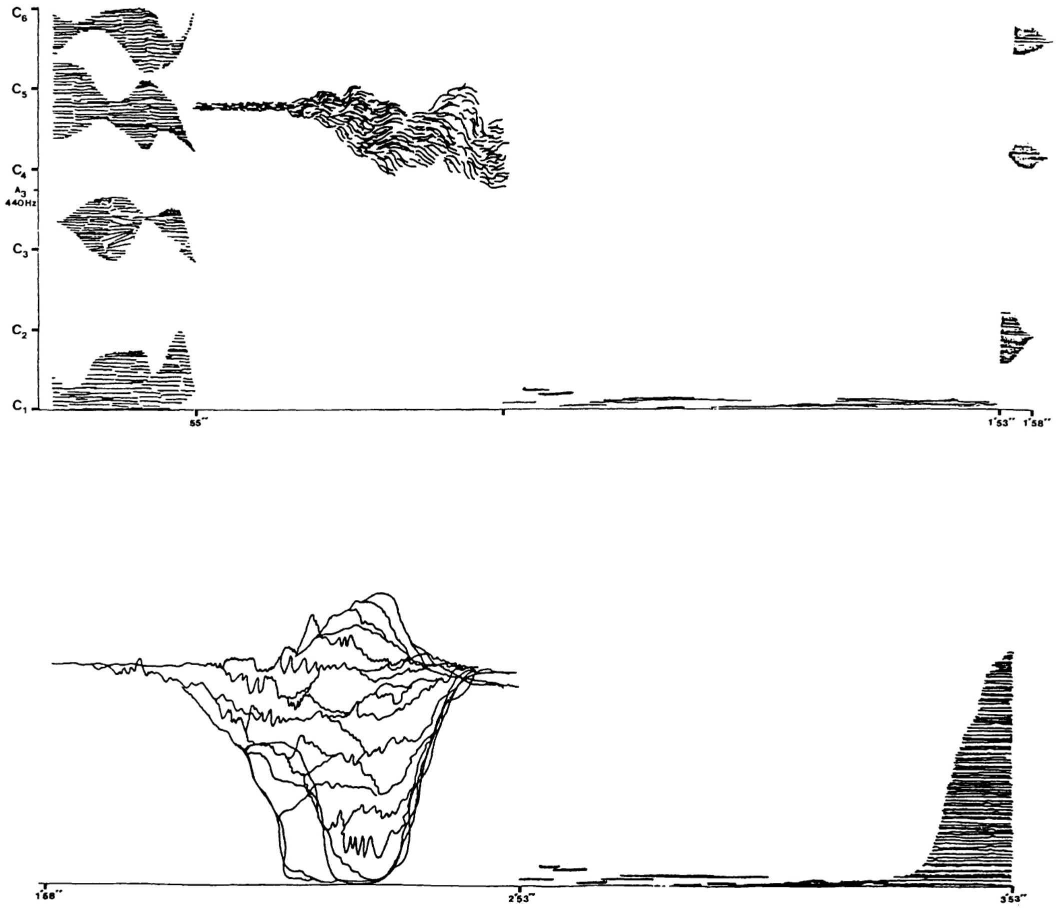


Figure 2.5: Iannis Xenakis - Mycenae Alpha score

give image source

waveforms rich in harmonics that cascade, flutter, crash, and scream like sirens in a vast cosmological territory" (Tyranny, 2017) .

The early version of the UPIC worked by drawing on a large digitizing graphics tablet which was interpreted by a high-powered computer (for that period) and converted into audio signals. The graphic approach to sound specification worked on a synthesis level by allowing the composer to draw and audition waveforms. Larger structures could be drawn in by switching to a "score" page and drawing lines, or "arcs" as they were denoted, on a pitch-time canvas. The final version of the application ran on personal computers and allowed for real-time interaction with a 64 oscillator synthesizer. At this stage, the input means had changed to a computer mouse but nevertheless retained the graphic approach of interaction. (Nunzio, 2014)

A primary goal of the UPIC project was that of pedagogy. Xenakis reasoned that the universality of sketching meant that it could provide an excellent teaching tool for a wide audience, even for young children (figure 2.6). Another goal of the system was

Figure 2.6: Iannis Xenakis showing UPIC to a younger audience

to encourage composer autonomy. At the time of its conception in the seventies, the technical barrier to entry into electronic music creation was very high and interfaces to help with this were rare or non-existent. Though the UPIC is not available to the general public currently, it has inspired a number of other systems that are available today. (Nunzio, 2014)

2.5.3 A Golan Levin's AVES

Golan Levin created the interactive audio-visual system, AVES, a series of audio visual installations in the late nineties and represented a landmark in the field of visual music. It is an attempt to move away from the diagrammatic approach to musical interfaces and to present an interface that is "painterly" in approach. Taking strong influence from visual artists such as Paul Klee, he presents a system that maps user input from a graphics tablet and mouse to visuals and audio. The intention is to create a strong visual correlation between these two modalities. A variety of approaches are taken to achieve this, all of them involving an algorithmic approach to a certain degree. For instance, in the piece "Aurora", he maps visuals of vast

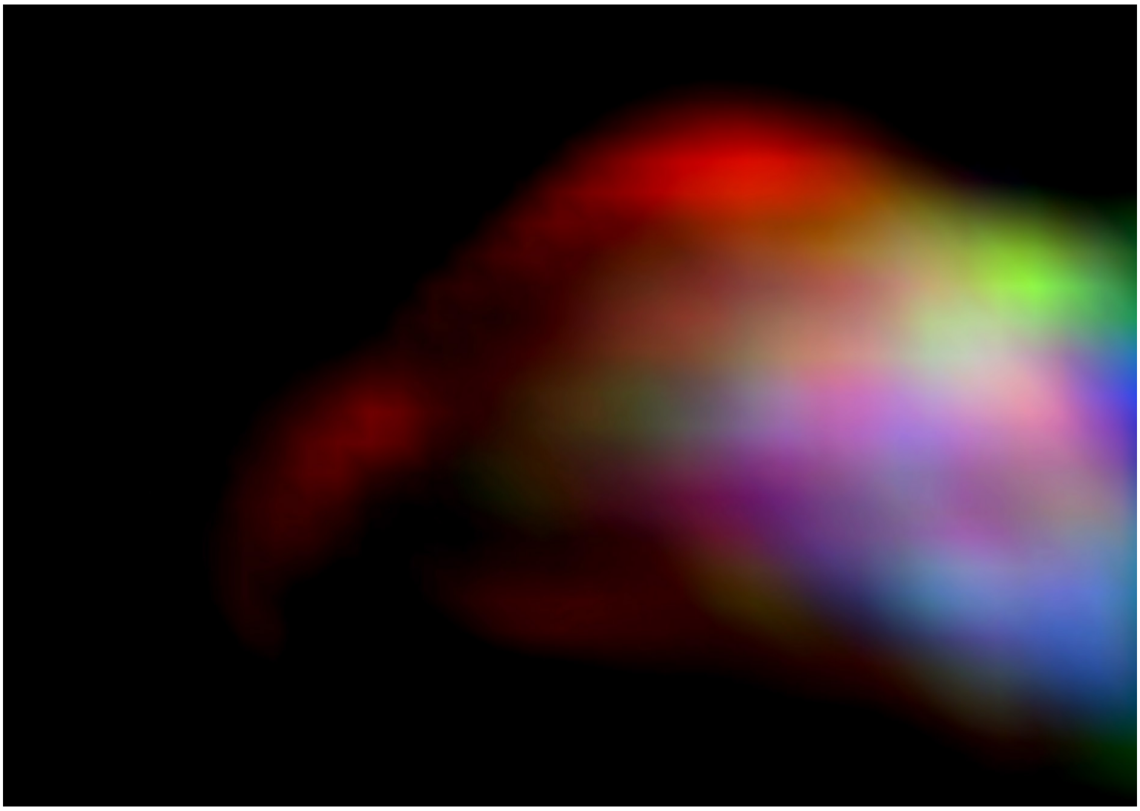


Figure 2.7: Golan Levin's Aurora (part of AVES)

quantities of particles to a granulated sound synth sound source. He didn't take the approach of an exact mapping of visual particles to audio particles, however, and instead used a statistical control approach to approximate the correlation between the visual and aural. (Levin, 2000)

For Levin, the digital pen input in combination with it's infinite variability represents an ideal instrument for creative expression in his digital temporal audio visual paintings. (Levin, 2000) The reason he gives for this is, similar to a musical instrument such as a violin, the pen is instantly knowable in that a child can pick it up and start creating marks but infinitely masterable through practice and hard work, and ultimately a vehicle for creative expression after a certain amount of mastery. A set of criteria that he and John Maeda arrived at to evaluate the success of their experiments was: is it instantly knowable, how long did you use it, how much of your personality can be expressed through it and, finally, with practice is it possible to improve using it. quote?

Levin's work is largely realtime and transitory in nature with gestures giving rise to visual and audio reactions that rise, fall and dissipate. A description that he uses of some of work is that of creating ripples in a pond. Therefore his work is very much geared towards an instrument-like experience. It is not concerned with the recording or visualization of a score or timeline of musical events as would be the function of a compositional tools such as a DAW. Indeed it is a conscious design decision to avoid such representations. Many of the principles and ideas of his work can, however, be applied in the context of a composition tool.

2.5.4 William Coleman's sonicPainter

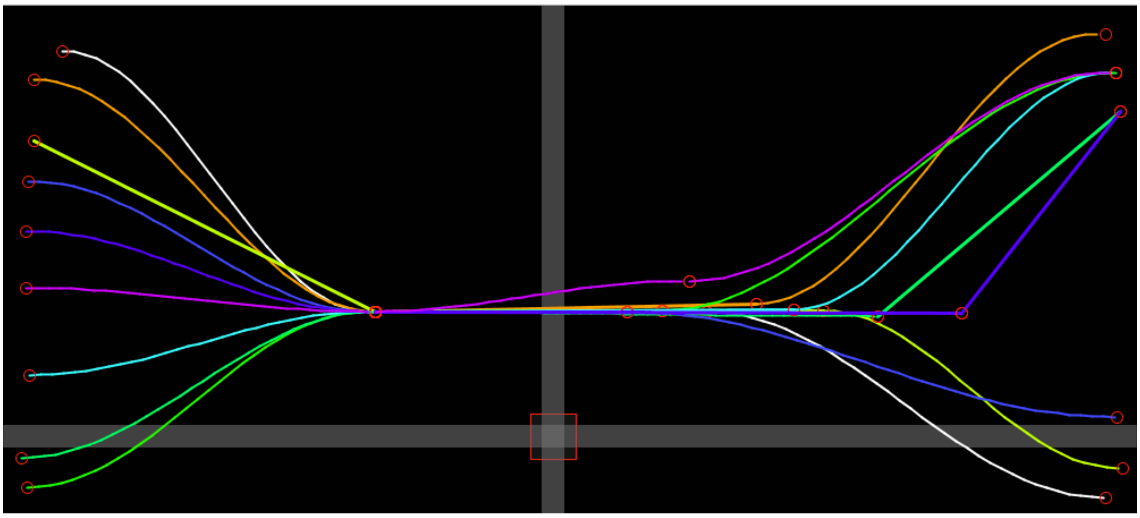


Figure 2.8: SonicPainter by William Coleman

SonicPainter by William Coleman is a novel musical sequencer that seeks to address some of the shortcomings of traditional approaches to music sequencing found in commercial DAWs (Coleman, 2015). The focus of the line and node based interface (see figure 2.8) is to bring timbral shaping to the fore rather than being hidden away in miscellaneous automation lanes. The design takes influence from legacy musical systems, in particular, the UPIC and incorporates ideas from visual music and embodied cognition.

Similarly to traditional sequencers, the x axis represents time and the y-axis, pitch. Note information is input via keyboard and mouse. A click starts a note and can be followed with additional clicks to continue to shape it. It can be ended by clicking a keyboard shortcut. By drawing notes as lines in this manner, the unfolding of the note can be explicitly represented visually. Other timbral aspects such as vibrato are represented by further visual manipulation of the line. For instance, an overlaid sine wave line indicates the timing and amplitude of the vibrato. In addition, the system allows for freehand input of notes.

2.6 Conclusion

The dominant metaphors present in DAWs, which are by and large analog studio influenced were discussed including details on their origins and their reincarnation in digital form. A short compositional example was given and the process to realise this in a DAW was described. The piano roll, multi-track mixer, and outboard effects metaphors were shown to be a poor fit for this particular compositional idea and resulted in an excessive amount of tracks and, therefore, complexity. A simpler solution was described in the csound audio programming environment. The lower level abstractions provided here allowed for a more succinct and simpler implementation of the piece. Some potential pitfalls to this approach were given. This includes a steep learning curve for novice users and a potential bias towards an analytical rather than a creative mode of thinking. Rather than abandoning the high-level metaphors present in DAWs it was posited that another approach could be to explore other metaphors more suited to certain compositional ideas. To this end, the metaphor of sketching as an interface to audio systems was explored by tracing it's early roots in the optical soundtracks of Oram to the realtime synth sketching of Xenakis's UPIC through to the contemporary approaches of Golan Levin's AVES system and William Coleman's SonicPainter.

Chapter 3

Methodology

3.1 Introduction

This chapter outlines the approach that will be taken in the realisation of the SonicSketch application. An assessment of SonicPainter will be given to identify the features that will be incorporated, improved on or omitted. The major technologies being used in the project will then be introduced along with some justifications for their usage. These include the *Web Audio API*, *Tone.js*, *Paper.js*, *ClojureScript* and Facebook's *React* framework.

3.2 Assessment of SonicPainter

As has been mentioned previously, the SonicPainter application developed by William Coleman will form the basis of the work on SonicSketch. While the basic design of SonicPainter works very well and is aligned with the goals of SonicSketch in the following ways:

- Provides a simple immersive score space that represents time on the x-axis and frequency on the y-axis.
- Allows multiple voices or timbres to be represented in this space so that all notes can be seen at a glance without having to click into separate tracks.
- Visualisation of vibrato effect using overlaid sine wave is suitable in theory but perhaps could be implemented better.

- Uses FM synthesis as the synthesis engine. This versatile and efficient is a good fit particularly for use in the web browser where performance can be an issue.

Some issues that have been identified with SonicPainter will now be addressed. It should be noted that as the software is only at a prototype stage, some of the shortcomings outlined here may not be design decisions and instead may be implementation issues.

3.2.1 Usability issues

Figures should have been provided to illustrate

While the application is conceptually straightforward and beginner friendly, there are some stumbling blocks that would make it difficult for a new user to begin using it. Starting a note is obvious and involves simply clicking on the screen to anchor the start point. A line now becomes attached to from this point to the mouse pointer location, indicating that another press will finish the note. However, this instead adds a point to the note, indicated by the fact that a new line appears between this new point and the pointer position. It is unclear how to finalise the note. The solution is to press the **shift** modifier key and click. Some simple labeling could alleviate this issue, by perhaps using a small toggleable information box. Unfortunately, this issue repeats itself in a good deal of the functionality with additional features hidden behind keyboard shortcuts that would be difficult or impossible to discover.

3.2.2 Issues with audio visual connection

Some deficiencies were found with the mapping between visuals and audio. It is possible to draw multiple concurrent notes that share the same timbre suggesting that the audio would polyphonically sound the notes together. This is not the case however and only one note can be played back at the same from the same timbre. To achieve this, a separate voice must be used by clicking the up arrow. To remedy this issue, it could either prevent the creation of concurrent notes visually or allow multiple notes to sound together by making each voice polyphonic.

3.2.3 Invalid states

A serious technical issue with SonicPainter is the presence of invalid application states that are not properly handled in the code. These arise when the user performs a series of actions that are not properly handled either by catering for them or by disallowing them. An example of this is when the user starts a note and then changes mode to, for instance, vibrato. When this happens, the application allows the mode to change and carry out the vibrato change all the while keeping a line connected to the pointer position. From here it is not possible to complete the note without returning to the original mode and clicking multiple times.

This unexpected behaviour could be avoided by either disallowing mode changes during note creation or by ending the note when the mode has changed. These type of issues are common even in commercial applications as is illustrated in Horrocks (1999). An example he gives is that of a calculator built by Microsoft that enters an invalid state in a number of different situations. To counter these issues a disciplined approach will be taken to managing state in SonicSketch by performing *validation* operations and the use of *finite-state Machines* (FSM). Validation is a technique used to ensure that data is in the correct format and that any required information is present and in an allowable format. A finite-state machine is a computational model that can be in only one of a finite number of states. These states may change due to external input which triggers transitions from one state to the next. They can prove useful in avoiding invalid states and to reduce the amount of conditional logic in the program.

3.2.4 Linear representation of frequency

SonicPainter represents a frequency range of 20-500 Hertz (Hz) on its Y-axis. This is mapped linearly from the on-screen coordinates to frequency values. It is not made clear why this range and mapping was chosen and may have been for aesthetic reasons. Psychoacoustically it may be suitable to map this linearly as this corresponds to the linear perception of frequency at the lower frequencies. This phenomenon is represented in the Bark Scale, which is a psychoacoustic frequency scale on which equal distances correspond to perceptually equal distances. However, the range of 20-500 Hz frequency is quite limiting given that the highest note of a piano can reach to 4186 Hz. In addition, there is a strong cultural expectation of logarithmic mapping of frequency owing to the semitone being a logarithmic scale.

3.3 Technical approach

A more general short intro to contemporary platforms would lead the reader into this discussion. Why were those below chosen out of all options? What are the functional features that necessitate their use?

3.3.1 Audio in the modern Web B

The W3C Web Audio API specification allows a in the web browser. Audio generation and processing is defined using an `AudioContext` graph of connected `AudioNode` objects. While javascript processing is supported (by using a `ScriptProcessorNode` object), most processing takes place in optimized lower level languages such as C or C++. Advanced synthesis techniques are possible by connecting audio generating nodes to processing nodes. Audio generating nodes include the `OscillatorNode` to generate a periodic waveform and the `AudioBuffer` to playback audio waveforms. Processing nodes include the `GainNode` to adjust the amplitude of a signal and `BiquadFilterNode` filter the signal. Parameters of these nodes may be adjusted smoothly over time using the `AudioParam` interface to, for instance, slowly fade a synth sound in.

3.3.2 Tone.js

Tone.js is a Web Audio framework that provides several helpful abstractions and libraries to help interaction with the Web Audio API. A central aim of Tone.js is to enable some of the conveniences of DAWs and is formed on three tenets: musicality, modularity, and synchronization. An example of this is the flexibility it allows to express time values, eg. “`4n`” for a quarter note in metrical notation, `1` for a second and “`100hz`” to express 100 hertz, etc. These are all converted to seconds before scheduling them with the Web Audio API. A system called “just in time scheduling” ensures that tempo-relative values are not scheduled until the latest possible moment, thus ensuring that they reflect the latest tempo value.

The primary reason that Tone.js was chosen was for the signal system that it uses to make working with parameter modulation easier and more efficient than working directly with Web Audio API. This uses a constant signal generator running at audio rate connected to a GainNode. In addition, this value can be multiplied and summed using GainNodes native capabilities enabling performant signal processing operations on all parameters. For instance, *FM synthesis* generally requires that a relative relationship is maintained between the two oscillators, the carrier, and the modulator. When directly using the Web Audio API Param, a `ScriptProcessorNode` would need to be setup to calculate this which is not as efficient or as straightforward as carrying out the calculations using Tone.js signals.

3.3.3 Paper.js

Paper.js is a descendant of Scriptographer, a scripting tool for Adobe Illustrator a vector graphics program. It runs in the web browser canvas element, running in its 2d mode. Paper.js adds a number of features to the browsers native canvas element including a *scene graph*, geometry and vector abstractions as well as tools to draw and animate shapes on-screen. The central abstraction in Paper.js and any vector system is the *path*. This allows for the specification of any shape by describing start points and endpoints for a series of paths. Curves can be added to these paths by adjusting an additional set of points associated with a path, the `handleIn` and `handleOut` points. These define control handles that alter the curvature of the line using Bezier mathematics.

A primary reason that Paper.js was chosen for SonicSketch is the path simplification algorithm that allows the data captured with freehand input to be simplified and smoothed. Instead of just plotting every point captured, an optimized subset of these points is used to reduce memory usage and speed up drawing. It is based on an algorithm by Philip J. Schneider published in Graphics Gems (1990). In addition, these simplified paths are more suited to mapping to control data for the audio system.

3.3.4 React framework

React is a web framework built by Facebook that aids the developer in updating the document object model (DOM), a process that is required when the state of the application changes. This was a role traditionally carried out by the web server and delivered to users as a static page. This saw a significant change however with the rise of single page applications (SPA) around 2010. The advantage of the SPA is increased interactivity and responsiveness to user input, allowing the look and contents of the page to update dynamically as the user interacts with the webpage. To aid in the construction of these SPA's a number of frameworks to help the process were introduced by the open-source community. Some popular early examples include Backbone.js and Angular.js. A technique that saw some popularity was a system called two-way binding which created a two-way link between the current state in the model and the visual appearance of the view. This, however, has a number of issues including some serious performance issues, in addition to some conceptual problems (Whelpley, 2014).

React offers a simpler one-way binding system using what is termed the *virtual DOM*. This works by maintaining a virtual version of the dom in a javascript. When the virtual DOM changes, the parts of the real DOM that require changing can ~~thusly~~ be pinpointed and efficiently updated. This system has proven to be particularly beneficial when paired with *functional programming* techniques, a style of programming that encourages the use of pure functions as the primary building block of programs. In the case of working with the DOM, it can lead to not only an increase in efficiency in the rendering of the applications but also a simplification of the programming model as a secondary benefit. A number of projects have emerged that attempt to bring this secondary benefit beyond the realm of the DOM. This includes writing command-line programs (Demedes, 2017), writing web audio applications (FormidableLabs, 2017) and even for embedded electronics (Kasten, 2017).

3.3.5 Clojurescript

Clojurescript is a compile-to-javascript programming language that is based on Clojure, a modern *Lisp* that runs on the Java virtual machine. Lisp is a programming language that was invented by John McCartney in the 1960's and is known for its minimal syntax consisting primarily of parens. The word Lisp is derived from the term "List Processor" as Lisp~~s~~ source code and data structures are built around lists. Clojure and ClojureScript promote a functional programming style. Clojurescript and other functional programming languages such as Elm have seen an increase in usage in the past number of years as this paradigm has proved useful in managing complex stateful UIs. Some annotated examples of ClojureScript follow that show the basic building blocks of the language and will help in understanding the code walkthrough in the next chapter.

```clojure
;; A list. It is quoted as otherwise it would treat it as a function call
'(1 2 3 4)

;; Call a function. In this case calling the multiply function.
(* 2 2)

;; Define a variable a
(def a 1)

;; Define a function b
(defn b [n] (* n n))

;; A vector is the most common way to represent sequential data in clojurescript
[1 2 3 4]

;; A hashmap is used for associative data and more often than not uses keywords
;; as keys. Keywords are used similarly to constant strings in other languages.
{:hello "world"}

;; Vectors and hashmaps are immutable and can't be changed after they've been created
;; An atom is used for data that needs to change (to model state changes for instance).
(def c (atom 1))

;; To change it's value use the reset! function.
;; Here it changes the c atom to 2.
(reset! c 2)

;; Clojurescript can interface with javascript in a straightforward manner
;; but uses a slightly different syntax. The term that is normally
;; used to describe this is js interop.
;; Calling a function:
(js/console.log "Hello world")
;; Calling a method on an object:
(.reload js/location)
```

This covers some basics that will be particularly helpful for readers that are familiar with C style languages. A more in depth is beyond the scope of this document and interested readers should consult the many resources available online. The most commonly used build tool that is used with ClojureScript (and Clojure) is *Leiningen*, which takes care of managing the code dependencies and converting the code from ClojureScript to javascript. These dependencies are defined in the "project.clj" file.

link

The primary dependencies used by SonicSketch, *Reagent* and *re-frame* will now be discussed.

3.3.6 Reagent & Re-frame

Reagent is a library that provides an idiomatic ClojureScript interface to React, allowing ClojureScript to harness the DOM manipulation facilities provided by React. This delegates the *side-effects* of rendering and manipulating DOM to React's reconciler algorithm. Side-effects is a functional programming term to denote anything that isn't related to the supplied arguments or return value of a function. This is normally object mutation (to change the state of the program) or input/output (I/O) operations, e.g. writing a file to disk, displaying graphics or playing a sound. (Sylwester, 2015) In addition to the interface to React, it provides a special reactive atom that efficiently re-renders React components when the state changes.

?

Re-frame is a framework that uses Reagent's interface to React to manage views and it's reactive atom to manage state. It proposes a program architecture consisting of the following 6 elements:

1. Event dispatch
2. Event handling
3. Effect handling
4. Query
5. View
6. DOM

The majority of events that are dispatched are due to user interactions with the system (for instance on a mouse click). Event handling is the code that is run in response to these events. Re-frame submits that these event handlers should supply data to describe the side-effects rather than carrying them out in the handlers. Re-frame carries out this work which is typically to update the application state. This is stored in a single reactive atom and managed by the framework. A subscription system allows the view system to update when the state that it depends on changes. Finally, React updates the DOM to complete the cycle.

3.4 Conclusion

SonicPainter was discussed in some depth with a focus on issues that will be improved on in SonicSketch. The technologies being used in the SonicSketch app were described in detail including some brief justification behind these technical choices. This covered the Web Audio API technology that underlies the audio aspect of the project, as well as the libraries and frameworks being used to manage the views and state of the app.

Chapter 4

Execution

4.1 Introduction

The following chapter gives an outline of the process that was undertaken to build out the final application. A description of early prototype work is given to give context to the construction of the final working prototype version. This is followed by a description of the technical architecture of the system as well as an outline of the sometimes difficult setup process of getting the various architectural elements working together. A detailed discussion of the core functionality of the system is then given, followed by a description of functionality that is more secondary in nature.

4.2 Early prototype work

4.2.1 MelodyPainter

MelodyPainter is an early protoype built ~~out~~ in Max MSP that allows users to draw freehand lines, which are converted into break point function data and used to generate a melodic profiles using Bach for Max MSP (Agostini and Ghisi, 2015). Bach is a suite of composition tools that allow for a number of computer aided composition techniques (CAC) and provides similar functionality to IRCAM's Open Music system. These melodic profiles are then filtered to only includes notes from a pentatonic scale, to give reasonably pleasing aural results. Some notable flaws in the system include the following. It is limited to strictly western tonal music styles. It

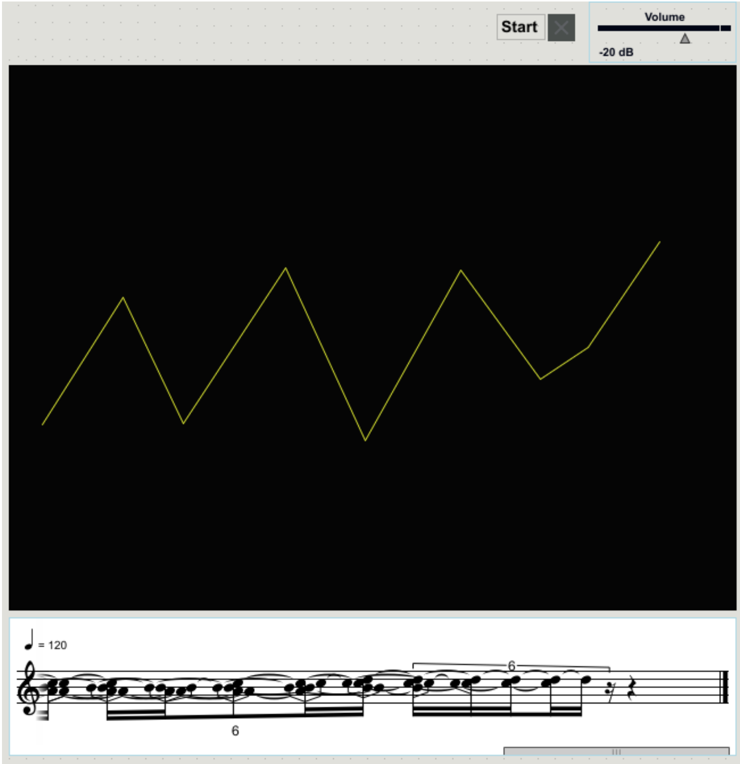


Figure 4.1: MelodySketch interface

has no allowance for rhythm and plays only eight notes giving results a noticeably bland and predictable quality. The freeform nature of sketched input however was quite a pleasing means of inputting the control information.

4.2.2 SonicShaper

A separate application was created in[1]Processing which allowed users to draw shapes, using either mouse or, ideally, pen input. A sound that is associated with each shape is then played back. As the sound of each shape plays back, it is illuminated using animation, creating a strong connection between the shape and it's resulting sound. The application uses the "gesture variation follower", a system that allows realtime recognition of gesture shapes (Caramiaux *et al.*, 2015). While promising in principle, it didn't have a high rate of accuracy, making it difficult to predictably control.

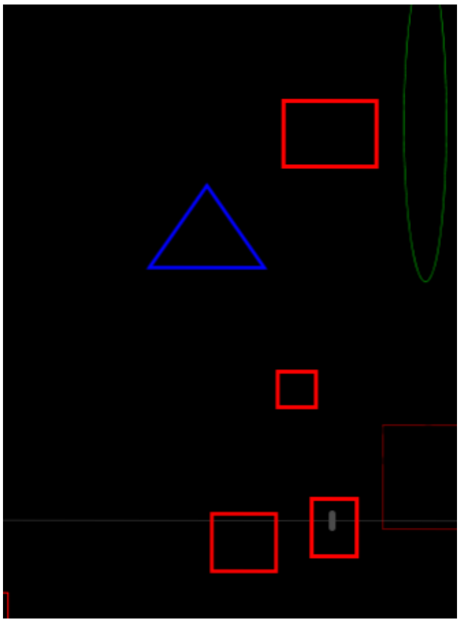


Figure 4.2: SonicShaper interface

4.2.3 Web version of William Coleman's SonicPainter

A potential starting point that was considered was using the code from ~~William's~~ SonicPainter and porting it to the web browser (Coleman, 2015). This process proved to be quite straightforward. The Processing code could be embedded in a webpage with minimal modification, using *Processing.js*, a web version of the Processing library that enables users to run Processing sketches in the Web Browser (Fry and Reas, 2017). Some notable changes that had to be made were removing the OSC functionality as this is not technically possible to use in a browser. In addition, some other pieces of code were removed and altered slightly. As it's not possible to run Max MSP patches in the browser, the audio system was re-implemented using Tone.js (Mann, 2015). As SonicPainter uses simple FM synthesis, a very close approximation to the original version could be created. In the end, it was decided not to build on this codebase however due to the issues with functionality and usability detailed earlier. These would be difficult to resolve in an inherited codebase. The process of porting the code did however give a more in depth incite into Coleman's implementation.

[1]Processing is a creative coding environment that runs in the Java virtual machine

There are some libraries e.g. osc.js

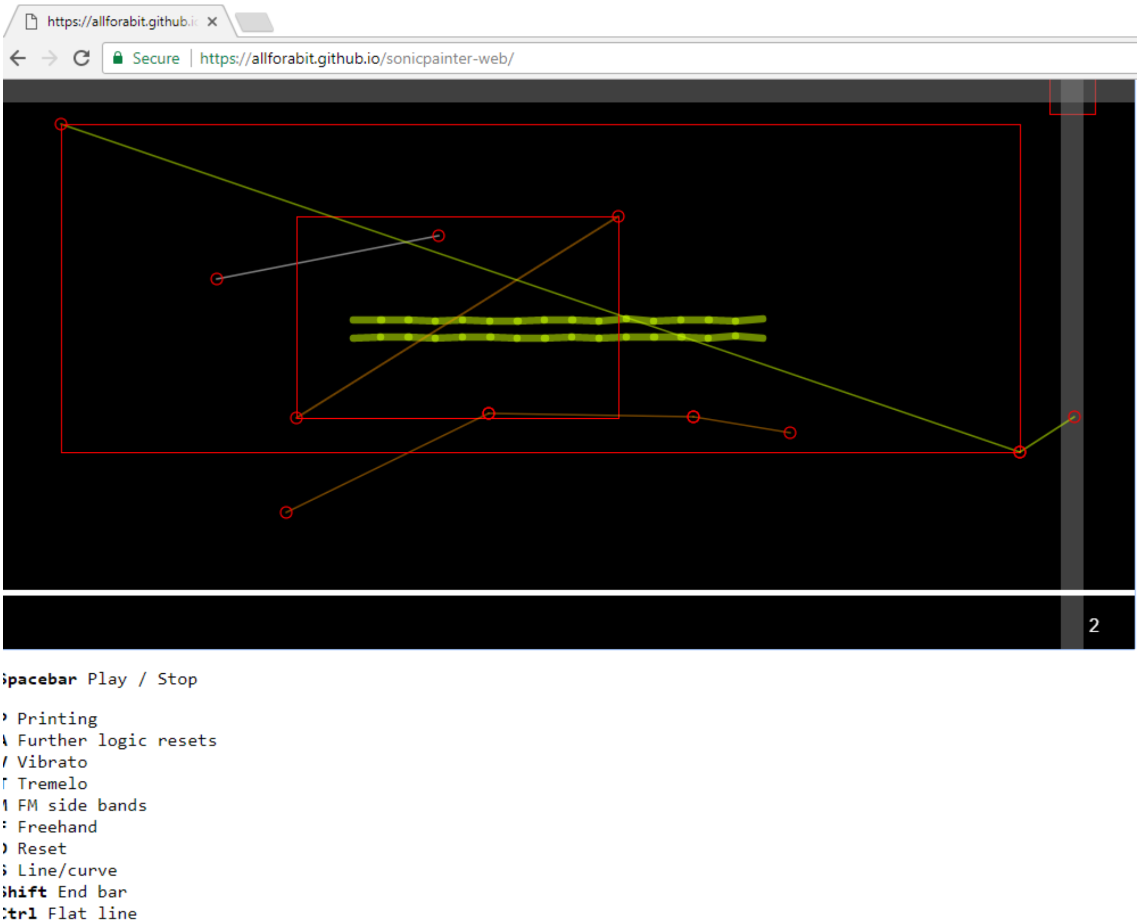


Figure 4.3: SonicPainter in a web browser

4.3 Setting up the architecture

4.3.1 Clojurescript and javascript npm modules

Despite the fact that clojurescript has existed for six years (Sierra, 2011), some areas of the development process are still difficult, particularly when building ~~out a~~ more complex real world applications. It should be noted that a good deal of work is being carried out to make this a smoother experience and, ~~thusly,~~ it is likely to become easier in the near future (Monteiro, 2017). It should also be noted that building applicatons using plain javascript is not a trivial process either and in will , in all likelihood, include a complex build process using a system like webpack or browserify. ?

A primary issue that had to be resolved to allow the application development to proceed was the incorporation of javascript npm modules. NPM is the package manager used by *node.js*. Node.js is a javascript platform originally designed for more server oriented applications, but, increasingly, also for rich client-side applications. The related NPM repository houses a large amount of javascript packages (currently 477,000) and is one of largest collections of code in the world . For a pure javascript application, it would be a matter of simply adding the desired libraries as NPM dependencies. However, with the use of ClojureScript, some extra steps needed to be carried out. In addition to adding these dependencies, a javascript file needed to be created that imported these into a `deps` object. This `deps` object could then be referred to in ClojureScript using the standard interop syntax `js/deps.myDependency` (Weller, 2017). At the time of development, an alpha feature that allowed npm dependencies to be declared as part of the project.clj file was experimented with but was not used due to some implementation difficulties. While the project setup was not as elegant or succinct as might be wished, it did provide a stable base to build on. Crucially the rich resource that is the NPM ecosystem could now be harnessed, to use such tools as Paper.js and React.js.

4.3.2 Paper.js and React (Paper.js bindings)

Paper.js runs in the context of a canvas element and ~~thusly~~ it is not possible to directly use React with it. This shortcoming has been addressed in projects such as *three.js react bindings* and *pixi.js react bindings* which allow the use of React's declarative programming style for 3d and 2d scene graph oriented systems that run in the html canvas element. These solutions both work by creating dummy empty DOM elements and hook into the *React.js* lifecycle events to do the real work of updating the scene graph. In many ways the scene graph structure of projects like these (and Paper.js) have a high resemblance to DOM structures and APIs, making React a good fit for them. A similar approach to the above mentioned libraries approach was taken to integrate Paper.js for use in SonicSketch. This worked reasonably well but required quite a bit of setup and ongoing development. During the course of the project build out, a more suitable solution emerged from the open source community. This used the next version of React (16), a version that has better support for render targets that are not the DOM. This has the distinct advantage of not requiring the creation of redundant DOM nodes. The library was far from comprehensive and~~, thusly,~~ a custom version of the library was used that included some custom functionality required for SonicSketch.

The relationships for all these elements would be better illustrated via a functional block diagram

4.3.3 Tone.js and React

In some ways audio output can be thought of in a similar way to the visual output of the app, merely as another type of I/O. ~~Thusly,~~ it can be treated in similar way by React and can use its declarative data oriented system to configure the particular settings and connections in the audio graph. React's lifecycle events can be used to instanciate the various audio generating and processing web audio nodes. This addresses a notable (by design) ommission in Tone.js which does not allow the state of the audio graph to be queried once it has been setup. It is the responsibility of the client code to keep track of and manage this. The advantage offered by introducing React into this part of the system is that it maintains the simple relationship between state and generated output. Conceptually the flow of change is:

1. The state updates
2. React components update their properties accordingly
3. React lifecycle events are triggered which take care of altering, adding and removing web audio nodes (thus altering the audio being output)

The design of this part of the application is influenced by *React Music*, a system that uses React with *tuna.js*, a web audio library similar to tone.js (FormidableLabs, 2017).

4.3.4 Reagent and React paper.js bindings

The final piece of the jigsaw in the underlying technology stack is the integration of React with ClojureScript via the *Reagent* library. The core syntax of this system uses simple ClojureScript vectors similar to the following:

```
[:div
 "Hello " [:span {:style {:font-weight bold}}
 "world"]]
```

This would result in the following html output:

```
<div>Hello <span style="font-weight: bold">world</span></div>
```

As can be seen, the vectors begin with a keyword that corresponds to the HTML tag name. Additionally, instead of using HTML tag keywords, function calls can be made to generate html by using symbols that reference functions. This allows for code reuse and logic. It was unclear how the Paper.js bindings would work within this system due to the fact that it required a different version of React and uses non standard tag names for the elements that can be drawn on screen such as “Circle” and “Rectangle”. This, however, turned out to be more straightforward than expected and the provided Paper.js primitives could be used by simply using the relevant keywords such as `:Circle` and `:Rectangle`. Complex scene graphs could be constructed by using the following succinct ClojureScript syntax to, for instance, describe the playback indicator:

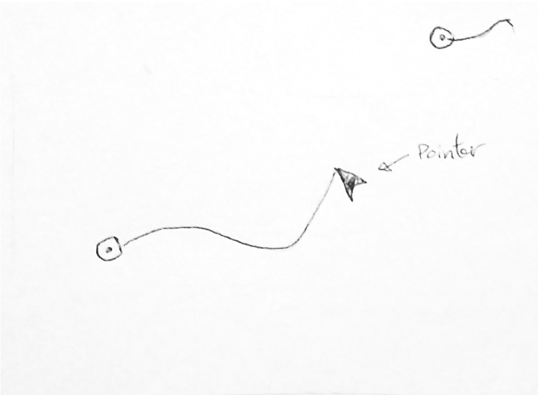
```
[:Group {:position [position 0]
         :pivot [0 0]
         :opacity    0.75}
   ;; This is the main bar that runs from top to bottom
   [:Rectangle {:pivot [0 0]
                :size [1 height]
                :fill-color "#ffffff"}]
   ;; This is the triangle at the top
   [:Path {:segments     [[-5 0] [5 0] [0 7] [-5 0]]
           :fill-color "#ffffff"}]]
```

The `position` and `height` are properties that are passed into the hiccup ? and trigger updates to the visual display when they change: in the case of position, when the playback position changes and in the case of the height, when the user resizes the browser window. The path element describes the triangle that is places at the top of the screen.

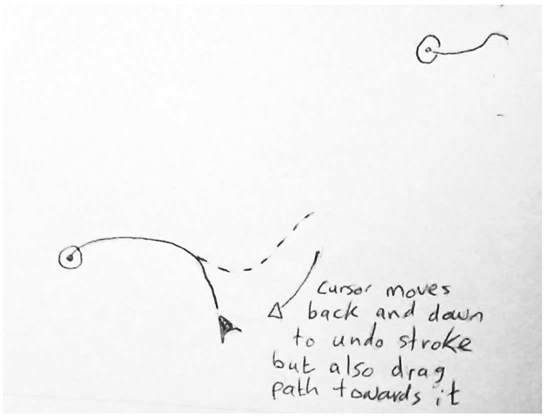
4.4 Core functionality - sketching notes

At the core of the application is the creation of timeline events which unfold in a looped fashion. These events are created based on the input of the user with a mouse or mouse-like input device. On the production of a valid input gesture, the screen is updated immediately with a visual display of this content. The details of this gesture is stored in memory and the event that will eventually create the sound is registered with Tone.js. Much of the events that occur in the system are captured in a main `View` component which houses the central html canvas element. To aid in organising the large amount of functionality associated with the component, higher order components are used to separate this out into logical groupings. A higher order component is a component that wraps a normal component to add functionality to it and accepts the same properties as the component it wraps (Facebook, 2017). In this case the most logical grouping is by tool and so there are higher order components setup for each of the tools: draw, vibrato, delete, move, resize and probability.

4.4.1 Adding notes



(a) User starts note by dragging left to right



(b) Dragging back clears note

Figure 4.4: Adding notes in SonicSketch

The sketch tool is the default tool that is activated when the user opens the application. It enables the user to add notes by drawing them onto the screen. The event is captured in the main canvas view and is initiated when the user left clicks the mouse to trigger the following function:

```
(defn pointer-down [{:keys [temp-obj active-preset]} evt]
  (let [pointer-point (.. evt -point)
        ;; Group circle and path are temporary shapes
        group         (js/paper.Group. (clj->js {:position
                        [(.. pointer-point -x) (.. pointer-point -y)]
                                                  :pivot [0 0]}))
        circle        (js/paper.Shape.Circle. (clj->js {:fillColor "#ffffff"
                                                         :radius    5}))
        path          (js/paper.Path. (clj->js {:strokeColor   "#ffffff"
                                                :strokeWidth   2
                                                :fullySelected true
                                                :segments      [[0 0]]}))]
    (.. group (addChildren #js [circle path]))
    (reset! temp-obj {:path   path
```

```
                  :circle circle
                  :group  group
                  :loc pointer-point})))
```

This function receives a hashmap with a reference to a ClojureScript atom to store the temporary visualisation of the newly created note. This function uses javascript interop to directly instanciate paper.js objects and add them to a shared group.

As the user continues to move the cursor further points are added to the path created in the `pointer-down` function. Some constraints however are placed on the creation of the path and only points that are past the last previous from left to right are added. If the users backtracks it lead to a deletion of points, providing an on-the-fly undo like behaviour.

```
(defn pointer-move [{:keys [temp-obj active-preset]} evt]
  (when-let [{:keys [path group] :as temp-obj} @temp-obj]
    (let [pointer-point (.. evt -point)
          rel-pos       (.. group (globalToLocal pointer-point))]
      ;; Only add positive points relative to first
      ;; Remove points greater than pointer-points
      (when-let [last-seg (.. path getLastSegment)]
        (let [first-seg   (.. path getFirstSegment)
              first-point (-> first-seg .-point)
              last-point  (-> last-seg .-point)
              pointer-x   (.-x rel-pos)
              amp-env     (-> active-preset :envelope)
              stage-width (.. evt -tool -view -viewSize -width)
              max-width   (if (= (-> amp-env :sustain) 0)
                            (let [time (+ (-> amp-env :attack)
                                          (-> amp-env :decay)
                                          (-> amp-env :release))]
                              (-> time
                                  ;; Seconds to beats
                                  (* (/ js/Tone.Transport.bpm.value 60))
                                  (time->euclidian stage-width)))
                            nil)]
          (when (or
                  (nil? max-width)
                  (< pointer-x (+ (.-x first-point) max-width)))
            (-> path (.add rel-pos))
            (let [greater-segs (filter
                                 #(> (-> % .-point .-x) pointer-x)
```

```
                              (.-segments path))]
                ;; Remove greater points
              (doseq [seg greater-segs]
                  (.removeSegment path (.-index seg)))))))))
```

Completion of a note occurs when the user releases the button and to trigger the `pointer-up` function:

```
(defn pointer-up [{:keys [temp-obj active-preset stage-size]} evt]
  (let [{:keys [path circle group loc] :as temp-obj} @temp-obj]
    (.simplify path 10)
    ;; Send the actual note
    (dispatch [:note-add (-> (path->note path loc stage-size)
                             (assoc ,,, :preset active-preset)
                             ;; Use the color from the active preset
                             (assoc ,,, :color (:color active-preset)))] )
    ;; Remove temporary objects
    (.remove path)
    (.remove group)
    (.remove circle))
  ;; Unset temp obj
  (reset! temp-obj nil))
```

This function simplifies the path by calling the paper.js `simplify` method on the path object and dramatically reduces the amount of data captured while preserving the basic characteristic of the user's stroke. Most importantly it calls the re-frame `dispatch` function to add the note to the `app-db`. A `path->note` function is used to convert the stroke from the domain of euclidean space in the visual space of the canvas to the domain of time-pitch space for use with the audio synthesis system. The `path->note` function can be seen below:

```
(defn path->note [path first-point stage-size]
  "Main entry point to this namespace"
  (let [path-width (.. path -bounds -width)
        width      (:width stage-size)
        height     (:height stage-size)]
    {:freq        (domain/euclidean->freq (.. first-point -y) height)
     :onset       (domain/euclidean->time (.. first-point -x) width)
     :duration    (domain/euclidean->time path-width width)
     :velocity    0.5
     :enabled     true
     :probability 1.0
```

```
    :color       @(col/as-css (get colors (rand-int 100)))
    :height      (.. path -bounds -height)
    :width       (.. path -bounds -width)
    :envelopes   {:frequency {:raw     (paper-path->vec path [width height])
                              :sampled (paper-path->sample path stage-size)}
                  :vibrato   (reduce (fn [a b]
                                       (assoc a b [b 0]))
                                     (sorted-map) (range 11))}}))
```

The domain of time-pitch is used to store the notes in memory and makes it possible to maintain a relative relationship between the screen size and the drawn notes.

The dispatched note event is handled by a re-frame `reg-event-db` handler which describes the alteration that is required to be made to the `app-db`. It also uses a series of interceptors, to perform validation of the `app-db` and to remove some of repeated code from the event handler functions. Interceptors are similar conceptually to middleware and is the place where all of the side-effects arising from an event are actioned. Moving application side-effects here ensures that they are isolated and the majority of the program can be kept as pure functions. As can be seen the handler function is very simple:

```
(reg-event-db
 :note-add
 note-interceptors
 (fn [notes [note-info]]
   (let [id    (allocate-next-id notes)
         note  (assoc note-info :id id)]
     (if (>= (:duration note) 0.001)
       (assoc notes id note)
       notes))))
```

This does a simple check to make sure that note has a minimum duration and if so alters the notes vector to include the new vector which will instruct re-frame to update the `app-db` with this new state.

The structure of the note hashmap is defined using *clojure.spec*, a core Clojure/Clojurescript library to perform data validation. The note specs are defined as follows:

```
(s/def ::id int?)
(s/def ::freq float?)
(s/def ::onset float?)
```

```
(s/def ::duration float?)
(s/def ::velocity float?)
(s/def ::color string?)
(s/def ::note (s/keys :req-un [::id ::freq ::onset ::duration]
                      :opt-un [::velocity]))
```

Although not specified here, notes also have an `envelopes` key that stores frequency and vibrato envelopes:

```
{:frequency {:raw [{:point [-0.01 99.7]
                    :handle-in [0 100]
                    :handle-out [0 99.8]}  ...
                   {:point [2.8 98.02]
                    :handle-in [-0.10 100.4]
                    :handle-out [0 100]}]
             :sampled [-0.36991368680641185
                       ;; ...
                       -2.172026174596564]}
 :vibrato {0 [0 0], ... 10 [10 0]}}
```

The update in state puts re-frame's subscription system into action and any views that are subscribed to the changed application state are now re-rendered. The `graphics-notes` view for instance is subscribed to `:notes`, `:graphics-stage-width`, `:graphics-stage-height` and `:mode`:

```
(defn graphics-notes []
  (let [notes         (subscribe [:notes])
        width         (subscribe [:graphics-stage-width])
        height        (subscribe [:graphics-stage-height])
        ....]
    (into [:Group]
          (map (fn [note]
                 ^{:key (:id note)} [graphics-note* ...]) @notes))))
```

When a note is added this render function will run which will call the `[graphics-note]` component for each of the notes, and update the visual display of the screen to show the new note. A similar process happens in the audio system except, in this case, new web audio nodes are created, timeline events are queued up at the correct time and audio envelopes are setup that trace the curve of the drawn lines.

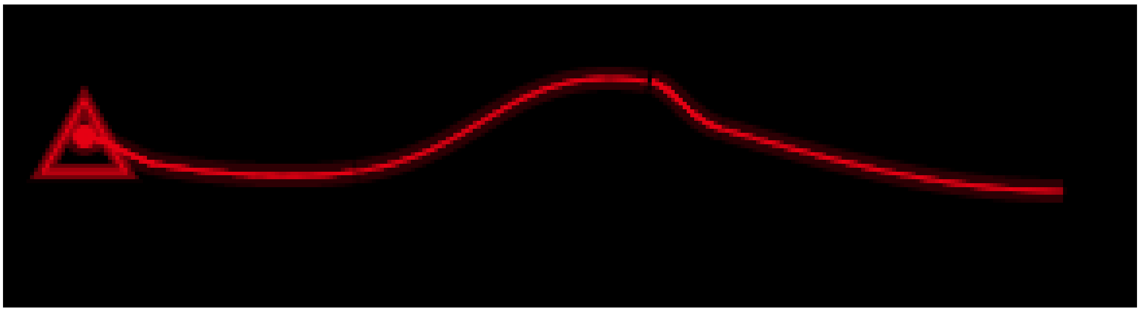


Figure 4.5: A note in SonicSketch

```
(defn fm-synth [{:keys [out] :as props} & children]
  "FM synth"
  ;; The new synth is instanciated using js interop
  (let [synth (js/Tone.FMSynth. (clj->js (dissoc props :out)))]
    (reagent/create-class
     {:component-did-mount
      (fn []
        ;; The synth is connected to it's output (which is passed
        ;; in as a property to the component)
        (.. synth (connect out)))
      :reagent-render
      (fn [props & children]
        ;; The render function renders a dummy span dom element and
        ;; renders it's children and passing it's synth as the out
        ;; for these components.
        (into [:span]
              (map (fn [child]
                     (assoc-in child [1 :out] synth))
                   children)))
      :component-will-unmount
      (fn []
        ;; Here we dispose of the synth
        ;; This will happen when the parent note is removed or when
        ;; a live code reload happens
        (.. synth dispose))})))
```

The above shows the `fm-synth` which sits at the heart of the audio generating part of the system. Potential parent components of this would be audio effects or the master bus. It's child components are comprised of note events and envelopes that drive frequency changes over the course of note playback. The composition of events, envelopes, synths, effects and channels is shown in truncated form:

```
;; Parent component is the project and
;; has settings such as tempo
[project {:project :settings}
 [master-bus {}
  ;; Master volume is set here
  [volume {:volume :settings}
   ;; Adds a simple reverb effect
   [reverb-effect {}
    [
     ;; First note
     ;; Each note has a vibrato effect
     [vibrato-effect {}
      ;; Envelope to control vibrato depth
      [timeline-evt evt
       [envelope {:param "depth"
                  :env   vib-env}]]
      ;; The fm synth that generates the signal
      [fm-synth (get-in evt [:preset])
       ;; Timeline event that takes care of queuing
       ;; it's child components
       [timeline-evt evt
        ;; In this case a note
        [note {:note :settings}]
        ;; And a frequency envelope
        [envelope {:param "frequency"
                   :state state
                   :env   freq-env}]]]]
     ;; Second note
     [vibrato-effect {}
      ;; ...
      ]
     ;; ...
     ]]]]]
```

4.4.2 Editing notes

Once created, a number of editing operations can be performed on notes. Apart from the **delete** action all of these operations involve first selecting the note on the **mouse down** event, carrying out the editing of the note and completing the action with the **mouse up** event. The particular tool can be activated by clicking the icon the right-hand side of the screen (see figure 4.6). Clicking one of these dispatches

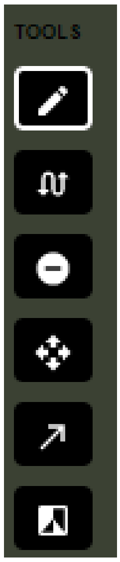


Figure 4.6: SonicSketch tools panel

the `:change-mode` event and changes the `app-db` to the specified mode as well as setting the application to state `:pending`. Flowing from this change are a number of changes:

1. The relevant paper.js tool is activated which will route further mouse events to the appropriate dispatch handlers. For instance, when the `:resize` mode is selected, **mouse move** events will be handled by `:resize-tool-pointer-move`.
2. It updates the visual look of the cursor to remind the user which of mode they are in.
3. A visual indicator is shown around the icon of the selected mode

The overall effect of this is an activation of a number of different modes, each of which will now be discussed.

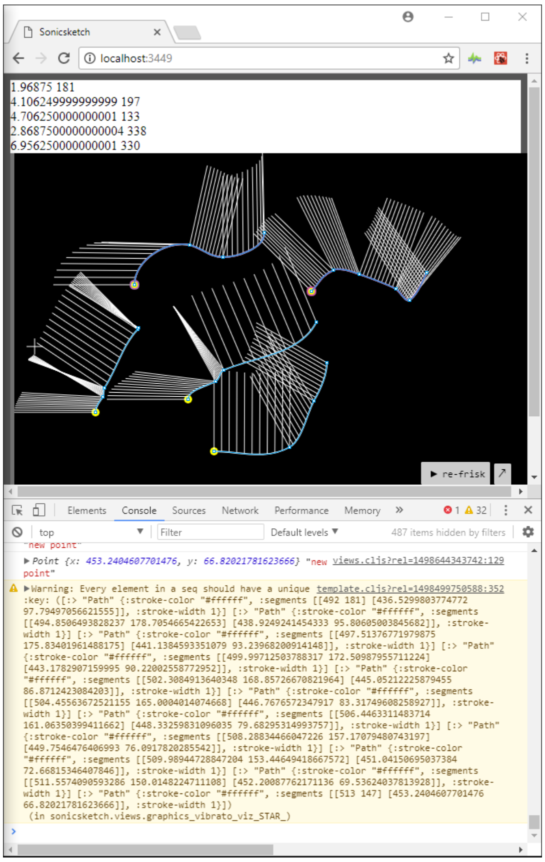
In the case of deleting notes, the event is simply raised in the **click** event of the note, which takes care of dispatching the `:note-delete` event. This is handled by a very simple handler, `(dissoc notes note-id)` that instructs re-frame to remove the note in question from the `app-db`. Similar to the process that occurs when a note is added, the visual display is now updated to no longer show the note. In addition, the signal generator, effects, envelopes and events associated with the note are removed from the audio component tree. React's lifecycle events take care of cleaning up any synths and effects by calling their `dispose` methods.

Moving notes and resizing notes work very similarly and both follow the most basic pattern described above of selection, manipulation, and completion. The note selection event is raised from the note's **mouse down** event handler, dispatching the `:note-select` event with the **note-id**. This updates the `app-db` to set the `:active-note-id` to the received id, sets the app state to `:active`, sets the note state to be `:active` and resets previously active notes to state, `:normal`. This state update changes the visual display of the active note to be highlighted by slightly lightening the colour of the note's surrounding glow. More importantly, the note now becomes the active item on which further user interactions are performed on. In the case of the **move** tool the stored onset and pitch properties of the note are altered to reflect the position of the user's mouse. This, in turn, updates the visual display of the note and causes the audio system to re-queue the note events to the new onset and pitch values. The **resize** tool, on the other hand, changes the size of the starting node of the note and alters the velocity accordingly. The altered size is calculated from the coordinates of the **mouse down** event and is clamped to a maximum size that corresponds to the maximum velocity of the note.

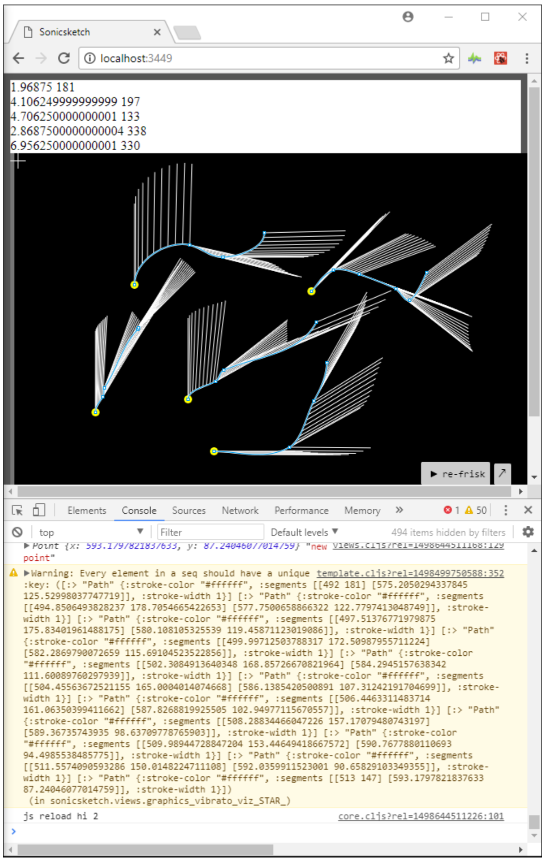
The slightly more esoteric **probability** tool works in a very similar fashion to the **resize** tools and again creates adjustments that work relative to the initial **mouse down** coordinates. The visual effect that this creates, however, is a dulling of the saturation of the note. Its effect is to add an element of randomness and depending on how saturated the note is, it will cause the note to randomly skip. The code for this is below:

```
(let [enabled (-> (Math.random)
                  (<= ,, probability))]
  ;; Only trigger the note if enabled is true.
  ;; If probability is 1.0 will always be true.
  (if enabled
    (do
      (rf/dispatch [:note-enable id])
      (some-> out
              (.triggerAttackRelease ,,, freq (prep-time dur) t velocity)))
    (rf/dispatch [:note-disable id])))
```

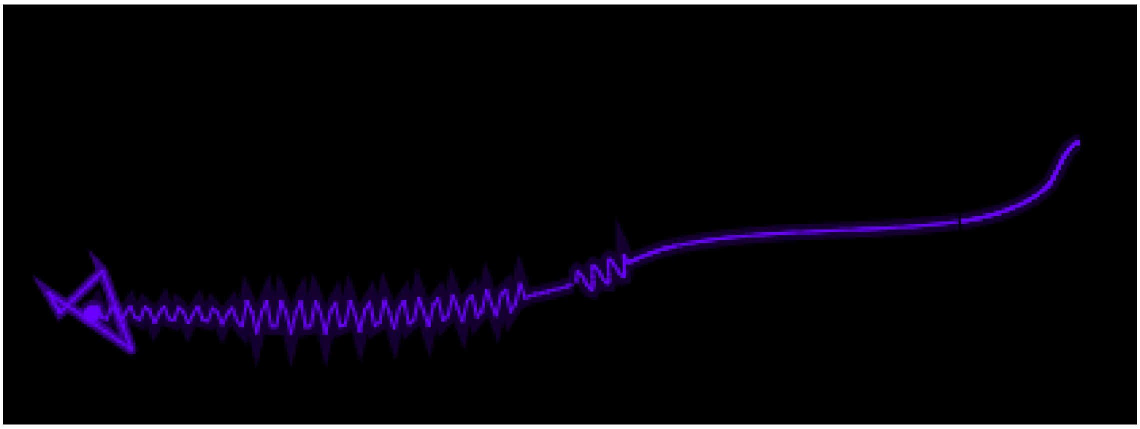
If the probability is fully saturated (corresponding to `1.0`, the default value), the note will play on every loop. If, however, it is below this it will skip in a random but probabilistic fashion, adding a small amount of stochasticism to playback.



(a) These lines show the normals of the curved line



(b) Lines showing the tangents of the curved line



(c) Completed vibrato visualisation

Figure 4.7: Live coding the vibrato visualisation

The vibrato tool is the most complex in its implementation and involves a small popup UI element that allows the user to draw in a vibrato envelope which is visually reflected in the note lines as an overlaid sinewave. After selection occurs, in the usual way, **mouse move** events dispatch to the `:vibrato-continue` handler which updates a vibrato modulation parameter in the note, a single float value that represents the current real-time vibrato value. This modulation parameter is passed back to the view through properties which triggers a `:component-did-update` function call. It is here that a dispatch is made to the `:vibrato-envelope-point` to create the envelope point. The reason for the back and forth between the view and the event handlers is to allow the event handlers to manage the state changes but deal with the specifics of the geometry of the vibrato overlay in the view.

A visual overlay is shown to the user when a vibrato action is started, that shows a 10 point envelope, whose points are all set to `0.0` by default. The user can then drag varying heights at various points on the horizontal axis and create the time-varying vibrato envelope. This envelope is visualised (and remains visible after the vibrato operation has completed) with a sinewave that tracks the curve of the frequency envelope and varies its amplitude depending on the strength of the vibrato at that particular point. The code to achieve this is below:

```
(->> (range 0.0 1.0 0.01)
     (map #(vector % (get-pos-of-curve-at-time curve %)))
     (map  (fn [[time p]]
             (let [norm          (.. curve (getNormalAt time))
                   loc           (.. curve (getLocationAtTime time))
                   relative-left (- (.. loc -point -x) left)
                   idx           (->
                                   ;; Normalize
                                   (/ relative-left width)
                                   ;; Mult by amp-env count
                                   (* (count amp-env))
                                   ;; Add amp-env starting point
                                   (+ (-> amp-env (first) (first)))
                                   ;; Round it off
                                   (Math/round))
                   amp           (-> (get amp-env idx)
                                     (second))]
               ;; Amp is here
               (sine-eq-along-vec time p norm (* amp 100) l)))))
```

4.5 Secondary functionality

Aside from the core functionality of note creation and editing, a number of other use cases were covered in the implementation. This includes transport controls, to allow the user to start and stop playback; some simple animation to show which notes are being played and to tighten the link between the visuals and the audio; undo and redo functionality; a fullscreen toggle; save and load functionality. Some of these extra features were made more straightforward than would normally be the case due to the architecture of placing the state in a single place, the re-frame **app db**.

Both the transport and the fullscreen mechanism consisted of a simple FSM whose transitions are caused by user actions. The transport state machine responds to both the **play toggle** button click and **spacebar** keypresses to toggle its state between playing and stopped and based on this, triggering playback in tone.js. The fullscreen FSM transitions occur not only when the user clicks the fullscreen button but also when the browser fullscreen status changes. This keeps the UI in a correct and consistent state at all times, regardless of how the state is reached.

Undo/redo and save/load functionality was relatively straightforward to implement for the reasons mentioned above (centrally located state) as well as the ease of serialising ClojureScript data. Adding undo/redo was as easy as adding an additional dependency and adding it as an interceptor to any events that needed to be undoable such as adding or deleting notes. The particular parts of the app state that needed to be restored were also configured and only included the `notes` collection and the `tempo`. The same two elements were saved and restored by the save/restore functionality which worked by serialising this data as a string and restoring with a single function call to *ClojureScript*'s `reader/read-string`.

Some simple animation was added to playback to increase heighten the connection between visuals and audio. To achieve this, the `:note` subscription is itself subscribed to `:playback-beat` a reactive value that is kept up to date via the use of events that fire on every animation frame. The subscription checks if the playback head is within the notes range and if so sets its `:playing` property to `true` and updates `:playback-time` to reflect the position of the playback head in the note. Similar to any other changes in state, the note views react to this and updates visual content accordingly to create the animation.

```
(rf/reg-sub
 :note
 :<- [:notes-raw]
 :<- [:playback-beat]
 (fn [[notes playback-time] [_ id]]
   (let [{:keys [onset duration] :as note} (get notes id)
         end-time                          (+ onset duration)]
     (let [updated-note (if (and
                              (> playback-time onset)
                              (< playback-time end-time))
                          (-> note
                              (assoc ,,, :playing true)
                              (assoc ,,, :playback-time (- playback-time onset)))
                          (if (true? (-> note :playing))
                            (assoc note :playing false)
                            note))]
       updated-note))))
```

4.6 Conclusion

This chapter outlined the development process that took place to bring SonicSketch to life. The early prototype work was detailed, in addition to the web port of SonicPainter, all work that contributed conceptually to the final artifact. The setup of the technical architecture that brings together tone.js, React, ClojureScript, Reagent and Paper.js was outlined. An in depth description of the primary functionality that enables users to draw and edit notes on the screen was given, followed by a more brief look at secondary functionality such as undo, saving and note animation.

Chapter 5

Evaluation

5.1 Introduction

An evaluation of the application will now be given. This is based on self-assessment, user feedback and a usability questionnaire that polled a small number of participants. User feedback was garnered at an exhibition that showcased the application. More informal ongoing feedback also identified some issues that were fixed in the final version of the application, in addition to some that weren't. Before delving into this discussion, let us briefly return to the composition mentioned earlier to see how it is realised in SonicSketch.

5.2 Returning to the composition

explain

As can be seen from the figure (figure 5.1), a more organic looking version of the Klangfarbenmelodie is in evidence. Depending, on the type of music being created this may be *too* organic and not accurate enough. Aside from these issues though, the visuals give a good visual representation of the underlying compositional idea. It is, in fact, very easy to imagine an idea of this kind originating in an environment like SonicSketch, whereas it is more difficult to imagine it originating in a DAW or an audio programming environment. Of course, equally there are other ideas that would be easy to execute in a DAW but impossible or very difficult in a graphic synthesis system such as this.

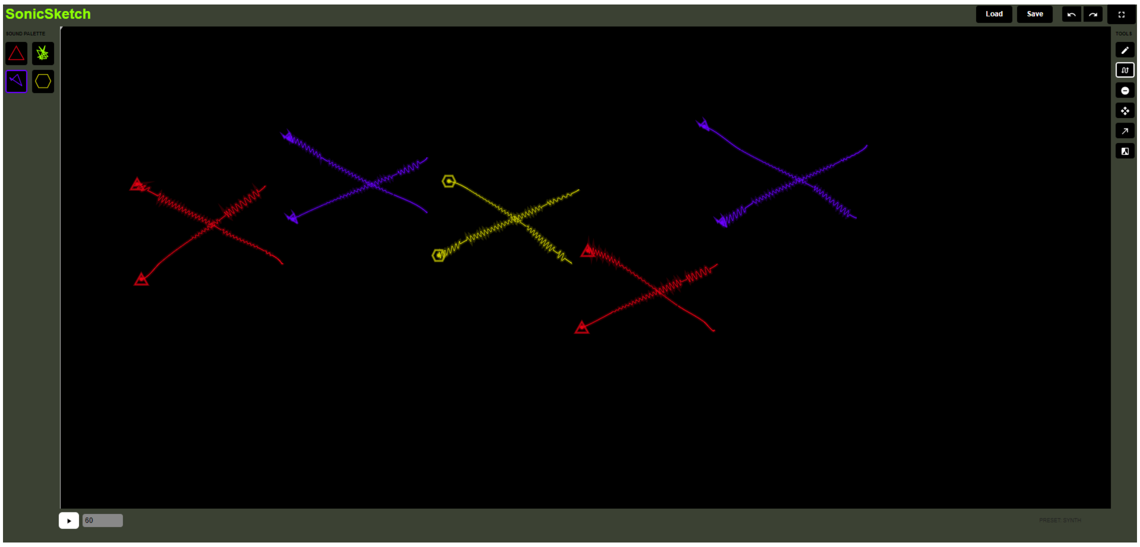


Figure 5.1: The Klangfarbenmelodie composition realised in SonicSketch

5.3 Assessment

5.3.1 System architecture

Overall, the chosen set of technologies worked very well and supported the technical requirements of the project. The use of recently released and beta versions of software libraries involved a good deal of time investment. Once this was setup, however, the live coding workflow enabled easy experimentation and proved invaluable when implementing complex features such as the vibrato overlay. The transparent data oriented approach of ClojureScript aided in debugging and made it straightforward to add advanced features such as undo/redo and save/load.

5.3.2 Core functionality - sketching notes

Adding notes was quite an easy concept for users to grasp and were observed to quickly start making marks on the screen. Some users experienced confusion or weren't aware of how to play these back initially. Once they managed to play the audio back and understood the general nature of the sounds being created, a common request was to clear the screen to start over. After this, they would start experimenting with layering up different sounds on top of each other. Oftentimes abstract visual patterns emerged and it seemed the visual feedback played as much, if not more, of a role than the audio, in what was drawn on the screen. Perhaps some real-time audio feedback might balance this out more. The version that was tested required that notes are drawn from left to right, causing some initial confusion for users. A future version will remedy this and allow for both directions. The woodblock timbre only allows short lines to be drawn as the sound isn't sustained and quickly decays after its initial attack. This proved to be another source of confusion for users and indicates that it may be inconsistent with other parts of the UI.

This is the type of design detail commonly described in Implementation

5.3.3 Timbre selection

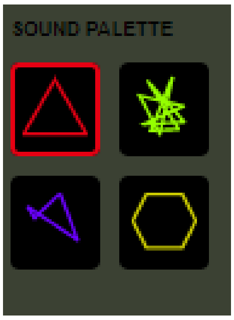


Figure 5.2: SonicSketch sound palete

Timbre selection is provided by clicking on one of the four coloured symbols to the left of the screen. The regular geometric shapes represent harmonic shapes and the more chaotic shapes represent non-harmonic sounds. Users did not seem to make this connection though. After sketching some notes and hearing the various timbres, most users were able to make the connection and to develop an intuitive relationship between the visual appearance and the audio result. Some users felt that the integration of timbre selection was a little flawed. When a tool other than the sketch tool was in use, and a timbre selection was made, the expectation was that it would return to the sketch tool. Instead, it stayed on the same tool but just changed the timbre. This will be improved on in future work. Another suggestion was to use icons that represent the instrument rather than the more abstract icons present in the UI. It was decided though that this would lead to an expectation of more realistic sounds and, thus, was avoided, however.

5.3.4 Vibrato

Similar to tools other than the default sketch tool, the vibrato tool was not heavily used by users and it was rarely discovered without some instruction. Once discovered though, most users were able to operate it after one or two attempts. The overlaid envelope graphic, while visually appealing in itself, doesn't fit the sketching metaphor very well and doesn't depart very far from what would be found in a traditional DAW. The resulting visualisation on the notes helps to tie it back in with the sketching theme. Ideally, the user should not see an overlay and instead directly adjust the visualisations on the note. Unfortunately, these visualisations weren't executed perfectly either and are applied slightly unevenly along the path of the note. It is a definite improvement on those present in SonicPainter though.

5.3.5 Other tools

Of the other available tools, delete and move were the most used. An initial usability issue was the accurate click required to target the particular note to be adjusted. Adding a glow effect to the notes remedied this to some degree and gave a slightly larger area to click on. Apart from this, there weren't any notable usability issues with either the delete or the move tools. The resize and probability tools were the least seldom used and when they were used, posed some difficulties for the user. They both worked on the principle of clicking on the note in question and then dragging in or out to alter the value. The drag motion didn't factor in the current value but was, rather, an absolute value defined by the point that the user clicked and the distance to the user's mouse pointer. Depending on the previous value, this caused a visual jump. In the case of the probability tool, the change in saturation is quite subtle and the changing colours may be difficult for the user to notice. The resize tool, which adjusted the velocity, only changed the node size and not the overall size of the graphic. This made it slightly unintuitive and confusing for the user, perhaps contributing its low usage.

5.3.6 Performance issues

The application requires a reasonably fast and modern computer to run in a usable state. However, even with fast hardware, complex heavily populated sketches lead to audio and animation glitches. Some optimisations were made to improve this. The primary adjustment was to add a `should-component-update` to the `graphic-note` component. This gets called by the React system to check if the component should re-render and is normally not needed for Reagent apps as it supplies a default version.

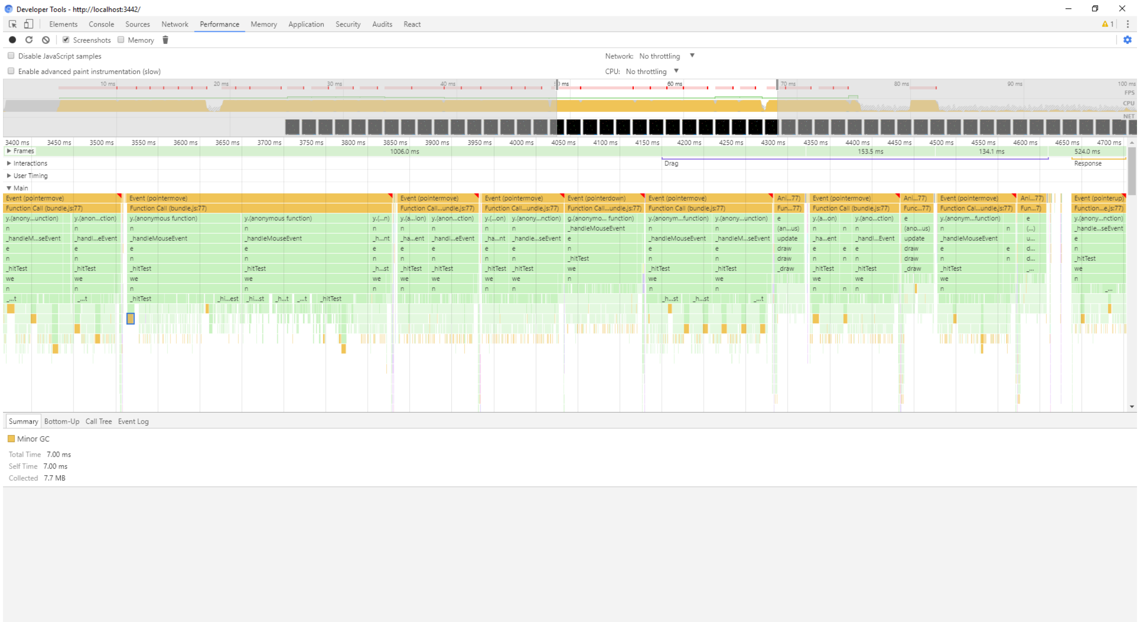


Figure 5.3: Performance debugging in Google Chrome

However based on performance testing with Google Chrome Developer tools it could be seen that a large amount of scripting was occurring on every frame (see figure fig:performance). This was the particularly the case when the note count went beyond a dozen or so. Introducing the custom `should-component-update` method reduced these renderings to an absolute minimum bringing a significant performance boost.

Even with this improvement performance is still an issue. This is largely due to the architectural decision to instantiate a new instrument for each note. Potential improvements to this could be to:

- Render content as audio at certain points by using an `AudioWebWorker`, a Web Audio API tool that allows rendering audio concurrently to a buffer. This would alleviate some of the burden on the CPU by decreasing the amount of audio generation it has to do.
- Use Web Assembly, a relatively recent Web technology that allows optimized C and C++ code to be compiled and run efficiently in the browser (Adenot, 2017). Current versions of both Csound and Faust (a functional audio programming language) can be compiled to run in the browser. The author was able to run a Csound version of John Chowning's "Stria" smoothly on a smartphone's web browser.

Future Work

- Use hardware accelerated graphics to further ease the burden of the CPU and improve visual display and animation.

5.4 User testing and feedback

5.4.1 Usability questionnaire

The System Usability Scale (SUS) is usability questionnaire that uses a Likert scale to give an indication of the how easy the application is to use. It poses a number of questions designed to provoke extreme responses either in favour of or against the proposition. Some examples of these are:

- I think that I would like to use this system frequently
- I found the system very cumbersome to use

The original introduction of the SUS questionnaire stated that the individual questions are meaningless and the results must be taken as a whole to give a unidimensional usability scoring (Brook, 1995). Lewis and Sauro (2009) have shown that this can be broken into two dimensions, however, usability and learnability. This helps to gauge how beginner friendly the application in addition to how generally usable it is.

Table 5.1: Results of SUS questionnaire

Factor	result
Learnability	92
Usability	83
Overall system satisfaction	85

The final score that the system got was 85 out of 100. Learnability scored higher than general usability, getting a total of 92 while general usability came in below that, with a score of 83. This gives a strong indication that the concepts and presentation of the app are easily grasped by novice users and that the general perceived usability is very high. Despite the fact that the number is scored out of 100 (with 100 being the highest score) it should not be interpreted as a percentage. Sauro (2011) has developed a grading system based on the results of over 500 tests, suggesting a grade of A to F, with A being the highest (figure fig:sus-grades). A score of 68 is average and would give a grade of C. Anything above 80.3 is an A grade and according to Sauro (2011), the point that users are more like to start sharing with family and friends. Therefore, by this metric, SonicSketch achieves a Grade A for overall system satisfaction, usability and for learnability.

Some caveats apply, however. The application was tested on a small group of participants (15), most of whom were quite familiar with working with audio and music applications. They were also colleagues of the author which may have caused a bias towards positive feedback. Nonetheless, the indication was that the application was straightforward and easy to start using, and provides a good basis for future work.

5.4.2 General feedback

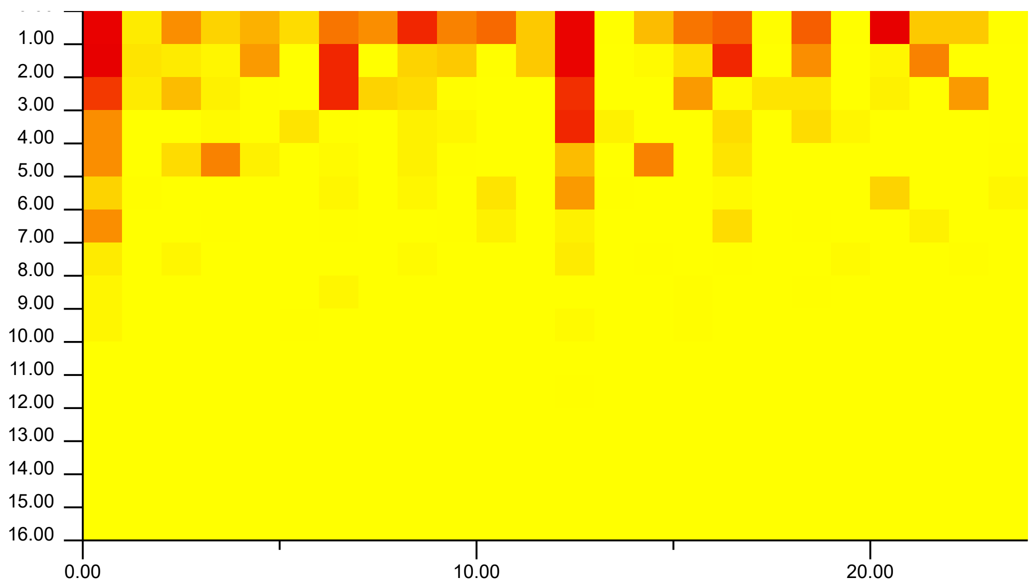


Figure 5.4: Heat graph displaying note start points

Overall the feedback from both the questionnaire and at the exhibition was positive with users reporting that it was a "fun and enjoyable experience" and that they "... could play with [it] for ages." A number of testers suggested that it would work well for sound design and cartoon sound effects in particular: "Really really fun! Can really see the benefit for sound design type scenarios, animation films and the like." Another recurring comment was that it would be interesting if you could sign your name and see what hear your sonic signature. Unfortunately, the app isn't able to give interesting results in this regard but it did show that users

Figure 5.5: Some exhibit participants managed to draw figurative artwork!

were engaging well with the concept of sonic sketching. Another user commented: "I enjoyed exploring how the different tools affected the audio and I liked trying to layer more and more sounds on top of one another." Again, this shows good engagement while at the same time pushing the prototype software slightly beyond its limits as it struggled to play back the ever increasing amount of audio material. A number of Sonic Sketches are presented that showcase the diverse approaches taken, with some users achieving figurative representations and another managing to (almost) sign her name (fig:user-sonic-sketches).

5.5 Conclusion

This chapter presented a critical assessment of the final application that factored in user testing and feedback. Each of the major features of the app were assessed in terms of their success in contributing to the overall application experience. As was discussed, some features worked well and didn't incur any major friction in usage whereas others leave room for improvement. Performance issues were discussed along with some potential remedies for these. Finally, the SUS usability survey was discussed along with general feedback received from users.

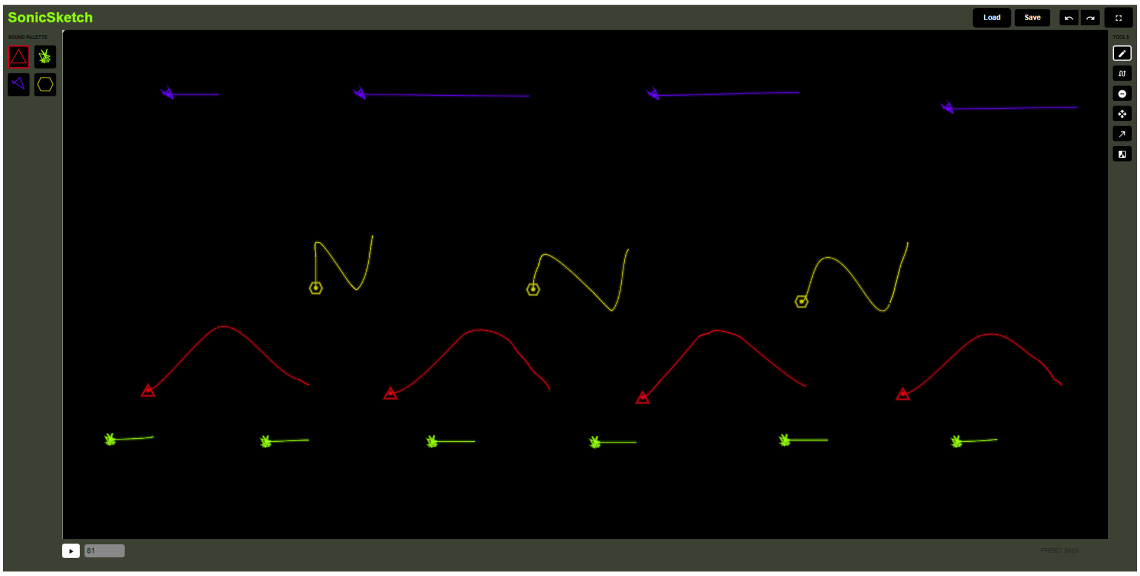


Figure 5.6: Some users explored visual rhythms

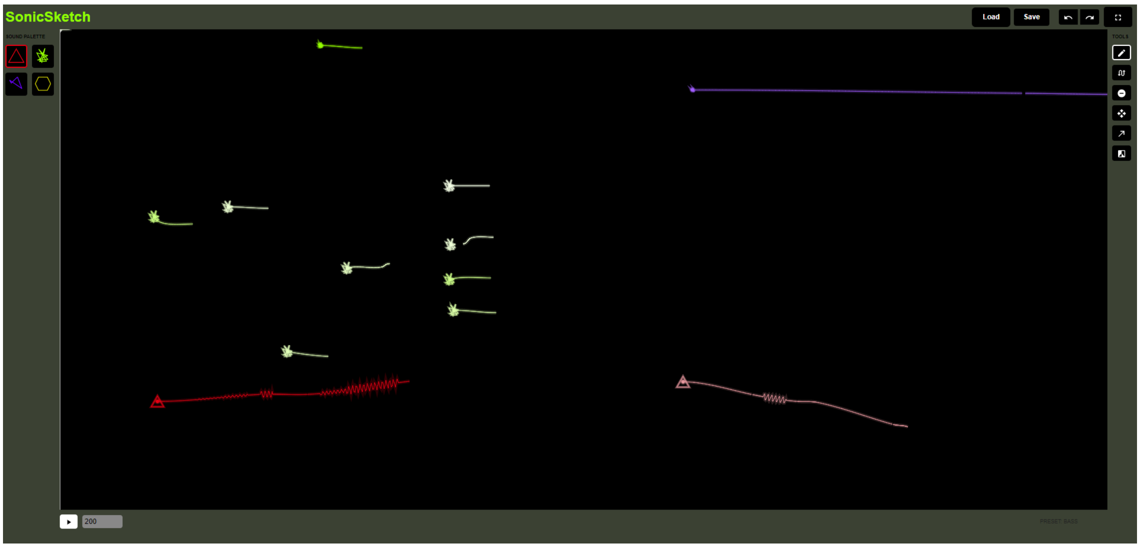


Figure 5.7: Vibrato added to a sketch

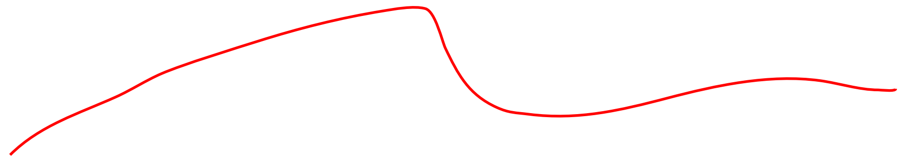


Figure 5.8: Exploring a clustered approach to sonic composition

Chapter 6

Conclusion and further work

This thesis presented some of the approaches that are open to modern musicians from the analog studio metaphors of the DAW to more open audio programming environments, to the diverse range of approaches to graphic synthesis. It presented SonicSketch, a prototype audio tool that runs in the modern web browser and allows musicians to sketch sonic ideas in on a time-pitch canvas.

6.1 Future work

There is an endless scope for improvements and features that could be made and the following represents some of these:

Performance improvements As is mentioned in the previous chapter, the application requires some performance improvements to make it a viable musician's tool. Ideally, the app should be able to run on mobile devices, as the uncluttered interface would be a good match for the limited screen size.

Larger structures The app doesn't offer any facility for larger structures. It would be interesting to explore how the sketching metaphor could translate to working at the arrangement level by perhaps introducing a level of recursion and allowing the user to sketch sketches.

Micro structures The preset timbres are a serious limitation to the current version. Taking inspiration from the UPIC, sonic flexibility could be greatly increased by allowing users to define timbres by sketching waveforms and envelopes.

Zooming and panning This would allow users to focus on particular parts of their sonic sketches and navigate through the sketch similarly to graphic editors like Photoshop.

More input devices Work was carried out to integrate graphics tablets into the application using the W3C Pointer API, a relatively new API that provides comprehensive support for multiple input devices including multitouch and graphics tablets (with pressure support and angle data). Unfortunately, it wasn't included in this version and is left for future work.

Grids While the freeform gridless canvas of the current version is fun to play with, an optional grid would allow for more precise control. Rather than only offering the chromatic grid, only suited to western music, however, a number of different grids could be made available. Perhaps a psychoacoustic grid based on the Bark scale could provide an interesting alternative or a grid based on Setharthes' dissonance curves.

The presented application represents a very small step in exploring the metaphor of sonic sketching. A virtually bottomless source of inspiration can be found in the many systems that have gone before it, some of which exist, and some of which can only be enjoyed through textual description, images and if, we're lucky videos. The modern web browser represents a unique opportunity to experiment with and develop these alternative approaches to music interface design in an environment that offers unprecedented ease of access for its users.

Bibliography

Ableton, L. (2017) *Live Set Export Documentation | Ableton*. Available at: https://ableton.github.io/export/ (Accessed: 17 August 2017).

Adenot, P. (2017) *Web Audio API performance and debugging notes*. Available at: https://padenot.github.io/web-audio-perf/ (Accessed: 4 August 2017).

Agostini, A. and Ghisi, D. (2015) 'A Max Library for Musical Notation and Computer-Aided Composition', *Computer Music Journal*, 39(2), pp. 11–27. doi: 10.1162/COMJ_a_00296.

Bell, A., Hein, E. and Ratcliffe, J. (2015) *Journal on the Art of Record Production » Beyond Skeuomorphism: The Evolution of Music Production Software User Interface Metaphors*. Available at: http://arpjournal.com/beyond-skeuomorphism-the-evolution-of-music-production-software-user-interface-metaphors-2/ (Accessed: 23 April 2017).

Brook, J. (1995) 'SUS - A quick and dirty usability scale'. Available at: https://pdfs.semanticscholar.org/c934/0d7584b8174df15f9296558c0441ca5ba045.pdf (Accessed: 22 August 2017).

Caramiaux, B. *et al.* (2015) 'Adaptive gesture recognition with variation estimation for interactive systems', *ACM Transactions on Interactive Intelligent Systems (TiiS)*,

excessive whitespace

4(4), p. 18. Available at: http://dl.acm.org/citation.cfm?id=2643204 (Accessed: 21 April 2017).

Coleman, W. (2015) *sonicPainter: Modifications to the Computer Music Sequencer Inspired by Legacy Composition Systems and Visual Art (MMT thesis).*

Cramer, A. (2002) 'Schoenberg's Klangfarbenmelodie: A Principle of Early Atonal Harmony', *Music Theory Spectrum*, 24(1), pp. 1–34. doi: 10.1525/mts.2002.24.1.1.

Demedes, V. (2017) *Vadimdemedes/ink: React for interactive command-line apps.* Available at: https://github.com/vadimdemedes/ink (Accessed: 23 August 2017).

Duignan, M. (2008) 'Computer Mediated Music Production: A Study of Abstraction and Activity'. Available at: http://researcharchive.vuw.ac.nz/handle/10063/590 (Accessed: 22 April 2017).

Duignan, M., Noble, J. and Biddle, R. (2010) 'Abstraction and activity in computer-mediated music production', *Computer Music Journal*, 34(4), pp. 22–33. Available at: http://www.mitpressjournals.org/doi/pdf/10.1162/COMJ_a_00023 (Accessed: 17 August 2017).

Facebook (2017) *Higher-Order Components - React.* Available at: https://facebook.github.io/react/docs/higher-order-components.html (Accessed: 23 August 2017).

FormidableLabs (2017) *React-music: Make beats with React!* Formidable. Available at: https://github.com/FormidableLabs/react-music.

Fry, B. and Reas, C. (2017) *Processing.Js.* Available at: http://processingjs.org/ (Accessed: 23 August 2017).

Horrocks, I. (1999) *Constructing the user interface with statecharts.* Harlow, England ; Reading, Mass: Addison-Wesley.

Kahneman, D. (2012) *Thinking, fast and slow.* London: Penguin Books.

Kasten, D. (2017) *React-hardware: A React renderer for Hardware.* Available at: https://github.com/iamdustan/react-hardware.

Levin, G. (2000) *Painterly interfaces for audiovisual performance.* Massachusetts Institute of Technology. Available at: https://dspace.mit.edu/handle/1721.1/61848 (Accessed: 23 April 2017).

Lewis, J. R. and Sauro, J. (2009) 'The factor structure of the system usability scale', in *International conference on human centered design.* Springer, pp. 94–103. Available at: https://link.springer.com/chapter/10.1007/978-3-642-02806-9_12 (Accessed: 22 August 2017).

Mann, Y. (2015) 'Interactive music with tone. js', in *Proceedings of the 1st annual Web Audio Conference*. Available at: http://wac.ircam.fr/pdf/wac15_submission_40.pdf (Accessed: 28 July 2017).

Monteiro, A. N. (2017) *ClojureScript - ClojureScript is not an Island: Integrating Node Modules*. Available at: https://clojurescript.org/news/2017-07-12-clojurescript-is-not-an-island-integrating-node-modules (Accessed: 23 August 2017).

Nunzio, A. D. (2014) *UPIC | musicainformatica.Org*. Available at: http://www.musicainformatica.org/topics/upic.php (Accessed: 18 August 2017).

Oram, D. (1972) *An individual note: Of music, sound and electronics*. London : New York: Galliard Ltd; Galaxy Music Corporation (A galliard paperback).

Overholt, D. (2009) 'The Musical Interface Technology Design Space', *Organised Sound*, 14(02), p. 217. doi: 10.1017/S1355771809000326.

Popova, M. (2012) *10 Rules for Students, Teachers, and Life by John Cage and Sister Corita Kent*, *Brain Pickings*. Available at: https://www.brainpickings.org/2012/08/10/10-rules-for-students-and-teachers-john-cage-corita-kent/ (Accessed: 2 May 2017).

Roads, C. (1996) *The computer music tutorial*. Cambridge, Mass: MIT Press.

Sauro, J. (2011) *MeasuringU: Measuring Usability with the System Usability Scale (SUS)*. Available at: https://measuringu.com/sus/ (Accessed: 22 August 2017).

Sierra, S. (2011) *Clojure :: Introducing ClojureScript*. Available at: http://clojure.com/blog/2011/07/22/introducing-clojurescript.html (Accessed: 23 August 2017).

Sylwester (2015) *The meaning of side-effect in Clojure - Stack Overflow*. Available at: https://stackoverflow.com/questions/31899630/the-meaning-of-side-effect-in-clojure (Accessed: 21 August 2017).

Tyranny, G. (2017) *Mycenae-Alpha (1978)/ in 'Electroacoustic Music: Classics' - Iannis Xenakis | Songs, Reviews, Credits*, *AllMusic*. Available at:

http://www.allmusic.com/album/mycenae-alpha-1978-in-electroacoustic-music-classics-mw0000877787 (Accessed: 18 August 2017).

Weller, T. (2017) *Clojurescript/Reagent : Importing React components from NPM*, *Tomer Weller / Blob*. Available at: http://blob.tomerweller.com/reagent-import-react-components-from-npm (Accessed: 23 August 2017).

Wessel, D. and Wright, M. (2017) '2001: Problems and Prospects for Intimate Musical Control of Computers', in *A NIME Reader*. Springer, pp. 15–27. Available at: http://link.springer.com/chapter/10.1007/978-3-319-47214-0_2 (Accessed: 16 August 2017).

Whelpley, J. (2014) *Is AngularJS Fast Enough?*, *Jeff Whelpley*. Available at: https://medium.com/@jeffwhelpley/is-angularjs-fast-enough-98dcf96406c8 (Accessed: 23 August 2017).

Appendices

Appendix A

SUS questionnaire

The following is the list of propositions that the tester must grade from strongly agree to strongly disagree.

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

Appendix B

Csound composition

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

kCarFreq = p4
kModFreq = p4*0.6666

kIndex = p6
kIndexM = 0
kMaxDev = kIndex*kModFreq
kMinDev = kIndexM*kModFreq
kVarDev = kMaxDev-kMinDev
kModAmp = kMinDev+kVarDev

aCarRaise linseg p4, p3, p5
aModRaise linseg p4*0.6666, p3, p5*0.6666
```

```
aVibAmpEnv linseg p7,p3,p8

aVibrato poscil aVibAmpEnv, 6, 1

aModulator poscil kModAmp, aCarRaise, 1
aCarrier poscil 0.3, aModRaise+aModulator+aVibrato, 1
outs aCarrier, aCarrier

endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1       ;Sine wave for table 1

; p1 p2 p3 p4   p5    p6 p7   p8
i 1  0  2  440  880   3  .01  20
i 1  0  2  880  440   3  20   0.01

i 1  2  2  660  1320  6  .01  20
i 1  2  2  1320 660   6  20   .01

i 1  4  2  330  660   2  .01  20
i 1  4  2  660  330   2  20   .01

i 1  6  2  880  1720  1  .01  20
i 1  6  2  1720 880   1  20   .01

i 1  8  2  220  440   9  .01  20
1 1  8  2  440  220   9  20   .01


</CsScore>
</CsoundSynthesizer>
```

Appendix C

Data disk index

1. sonicsketch-src - source code for application
2. sonicsketch-app - single page web application (open index.html in chrome browser)
3. kevin-nolan-sonicsketch-thesis.pdf - soft copy of thesis