University of Dublin

TRINITY COLLEGE

Creating a Collaborative Environment for Live Music Composition Using Sonic Pi

James Mulcahy

B.A. (Mod.) Computer Science

Final Year Project May 2018

Supervisor: Dr. Glenn Strong

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

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

James Mulcahy

Date

Permission to lend

I agree that the Library and other agents of the College may lend or copy this report upon request.

James Mulcahy

Date

Abstract

Emerging at the turn of the millennia, Live Coding is a field where computing and music meet. Performers create music through code which can be constructed and reconstructed during runtime. In recent years as this relatively small field has become more popular and there was been an increase in research and development of creating collaborative environments to support multi-user Live Coding.

This paper presents a collaborative environment for the Live Coding environment, Sonic Pi, in the form of collaborative control, where users connect their mobile devices via a server to Sonic Pi allowing them to jam together.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Glenn Strong, for his help throughout the course of this project. He was always available to help answer my questions and issues as well as providing great advice and guidance throughout the course of this project.

I would like to thank my parents for their continued support and encouragement.

Lastly, I would like to thank all the friends I have made throughout my time at Trinity as well as TrinityFM for always being a stress-free and inviting home on campus.

Contents

Chapter 1

Introduction

1.1 Introduction

Sonic Pi is a live coding environment based on Ruby, developed by Dr. Sam Aaron at Cambridge University in collaboration with the Raspberry Pi Foundation, a charity founded to promote computer learning in schools [1]. This software was designed as a tool to support computing and music lessons in schools by enabling students to code the kinds of music that they were listening to and thereby learn the fundamentals of programming code. The software allows one to make musical patterns with text, describing sequences and ways of transforming and combining them, exploring complex interactions between simple parts. In addition to the traditional stop start model of programming Sonic Pi allows the user to continually tweak and modify the program as it runs.

This project extends the Sonic Pi environment by allowing two or more users to remotely collaborate on live music performances enhancing the usability and creativity of Sonic Pi. Users connect their mobile device via the application TouchOSC to a node.js server, which manages step data and passes the information it to Sonic Pi which interprets it and provides playback. This involves looking at number of difficult issues including synchronization between devices and Sonic Pi which may have potential applications in other areas. In a world where young people are accustomed to communicating online, the collaborative environment outlined in this project is not only reflective of how young people interact with each other today, but will by providing an interactive, fun enjoyable experience encourage more young people to become curious about not only Sonic Pi but computing in general.

The collaborative environment created in this project provides a completely new experience, where multiple users can now form live musical groups performing on different instrumentation simultaneously, to produce live compositions. Visually seeing on their device as well as hearing the input of other users jamming with them in real time. This project showcases one implementation of live simultaneously composition as well as providing the software groundwork to allow an almost infinite array of setups and designs. From duo DJ collaborations to an orchestral type collaboration with multiple users, there is a wide area for development that can be built off this project's infrastructure.

1.2 Structure of Report

Chapter 2 briefly discusses the history of the field of live coding before getting into the current research and development of collaborative composition within live coding and discusses the current methods of approach as well as the issues that arise when creating collaborative environments.

Chapter 3 is broken into two main parts, "Exploration and Experimentation" which explores and examines the different approaches this project could have taken towards design, and "Design Architecture" which discusses the design implementation undertook based on the exploration and examination findings.

Chapter 4 contains a detailed description of the development and implementation of the project.

Chapter 5 outlines the limitations as well as possible improvements and future development of the work undertaken here.

Chapter 6 presents the author's conclusion

Chapter 2

Literature Review

2.1 Brief History of Live Coding

Firstly, a clear distinction must be made between “musical live coding”, where a program is edited while it is running live and an idea of “live coding”, where a program is developed in front of an audience like one might see as part of a lecture. The latter is not reprogrammed at runtime while the former is. Within the context of this report I will be referring to “musical live coding” simply as live coding.

Live coding is still a developing field with the first publication written in 2003 [2]. Coupled with the first meeting of changing grammars, at the University of Fine Arts Of Hamburg in 2004. This conference provided a decisive moment for the form live coding has today [3]. It showed how the idea of being able to rewrite programs at runtime wasn’t just relevant to a few individuals, but a problem with many aspects and applications which could be approached with many unique solutions [4] [5]. The implications of having so many unique solutions would later be discussed in depth, with many developers highlighting the issues caused by having so many unique solutions. Furthermore, it helped cement two important definitions of the word “live” which still remain central today [6]:

1. Programming as public thought
2. Reprogramming a program at runtime.

Programming as public thought refers to the way in which live coding is to be expressed and observed. From a large concert, to a classroom, or even an abstract concept. Reprogramming a program at runtime refers to one of the key functionalities of live coding, which is the ability to change and manipulate sounds in real time.

The development of live coding since that first meeting in Hamburg has in many ways gone beyond the expectations of that time. Resulting in the founding of TOPLAP [6], an organization to explore and promote live coding. It provided a central hub for the expansion of the field to a broader community and an open definition of the practice and its terminology [7]. This has led to the birth of many live coding environments such as ChucK, Extempore,Impromptu, Overtone, Supercollider, ixi Lang, Tidal Cycles and Sonic Pi. This increasing popularity for live coding is resulting in a heightened interest and awareness of collaborative composition as another way to express live coding and bring it to the next level.

2.2 Collaborative Composition Overview

The first instance of a collaborative real-time editor was demonstrated by Douglas Engelbart in 1968 as part of what is commonly known as “The Mother of All Demos” [8]. However widely available implementations of the concept would take decades to appear. Beginning with instant update for Mac OS in 1996 and more recently, thanks to Web 2.0, Google Doc in 2006.

Sonic Pi was initially released in 2012 and since then has seen increased popularity due in part to its simplified design and nature. It is still in development and as such the focus on development of a collaborative environment for live composition is not of immediate concern to the lead developers. However, collaborative live coding composition is currently a popular area of research within the live coding field, with external applications such as Troop [9] and Extramuros [10] being developed to support multi-user sessions. These applications use collaborative editing to solve the problem of collaborative composition, where performers use a shared text buffer.

The issues that must be taken into consideration while building a live coding environment, “time, latency and sample rate” [4], are still a central concern while extending an environment to allow for collaboration. As mentioned earlier, live coding environments and systems often contain quite unique solutions to the problem of how musical events are scheduled in time. For example, the live coding environment, ixi lang uses characters to represent events and the space between them to represent a non-event or silence. This leads to the code having a “graphical, spatial, and score-like representation of the music” [4].

This approach of unique solutions across environment may cause a number of issues to arise. There is scope for very different solutions to exist that are nothing like the ones discussed here; it's not clear that one solution to the problem of collaboration in live coding will comprehensively address a similar problem in another live coding environment. While this represents an obstacle in the future development for live coding environments, it is also what makes the field of research so interesting, having to solve problems of human-machine interaction, musical composition and performance.

As well as synchronizing and timing, when it comes to collaboration, data sharing is an important feature as pointed out by Andrew Sorensen and Andrew R. Brown [10]. Open Sound Control (OSC) has become a de-facto standard for musical communication. Open Sound Control or OSC for short is a digital media content format for streams of real-time audio control messages [11]. OSC was originally developed, and continues to be a subject of ongoing research at UC Berkeley Center for New Music and Audio Technology (CNMAT) [12].

OSC supports a number of live coding languages such as Sonic Pi and Impromptu, allowing for communication between a variety of other computer music tools supporting the OSC protocol. OSC implementation in Sonic Pi to allow for external device communication is new, only coming into effect in the last major version update. However, internally OSC has been used from the beginning to allow different parts of Sonic Pi's architecture to communicate to each other see §3.3.2 Sonic Pi. OSC can be used in live coding languages to allow the user to connect a MIDI device such as a keyboard to help with creating a more fluid live performance. More information on OSC in §3.2.3 OSC

The issue of setting up a shared temporal frame to create a collaborative environment came up in a seminar led by David Ogborn in McMaster University [5]. Ogborn argued that "The infrastructural issue is not a lack of ways of sharing timing information, but rather an excess of such methods." [5] Explaining that a lack of agreed upon protocols around synchronization techniques is what will really hamper collaborations, especially in the case of where multiple, different live coding software environments are used. Ogborn therefore concluded that it is optimal to have "synchronization elements independently of those performance environments" [5] to allow for the synchronization techniques to be tested and changed independently of the code actions carried out by an individual performer. This method of having synchronization separate to the live coding environment agrees with Thor Magnusson argument about the nature of live coding environments [4]. Due to the many unique solutions that exist within live coding environments, if one was to code the collaboration and synchronization into just one environment it is unclear whether it would be able to address the problem in another environment. The idea of synchronizing separately is key part of the design element of my project, see §3.2.4 independent synchronization.

2.3 Methods of Collaboration

Performing together from multiple computers or mobile devices can provide a challenge from an ensemble point of view. Synchronization, data sharing and timing must all be managed in order for a multiple user performance to work. Live coding performance groups have often considered using shared network resources such as a tempo clock to coordinate rhythmic data [13] to manage this. However, there is still no live coding collaboration standard as new methods and approaches to address the problem are continuously being developed.

In a mash up of Google docs meets live coding, a novel collaborative editing tool, Troop is being developed [14]. Designed as part of a PhD thesis exploring collaborative tools and processes in live coding, Troop showcases a different way to solve the problem of collaboration. The application allows users to work simultaneously on the same piece of code from multiple machines [8]. Built to be used for the Live Coding language FoxDot [15], Troop provides a shared text buffer that performers can see and contribute to at the same time. This approach to collaboration is called collaborative editing and is unusual as in the past rarely have performers worked together with the same material. Troop highlights one possible solution to the problem of collaboration in live coding.

Live coding ensembles will often coordinate their performances over a network to allow for the synchronization of their music and sharing of information between each user. This introduces several challenges for both the performers and audience members during a live show. As mentioned previously, one of the main technical issues facing a live coding ensemble is temporal synchronization, which brings forth the issue of latency. The amount of time it takes a message being sent from one device to be received by another, must be adjusted for when dealing with multiple computer clocks.

One of the approaches resolve this issue while performing over a large geographical distance is to have the synchronization of all parts be completed at the end connection. This technique of synchronization was used by Ogborn in his live demonstration, “Very Long Cat”, where he synchronized live coding and live percussion over the internet [16]. Ogborn used an application called JackTrip [17] to establish an audio link between his location, McMaster University’s Department of Communication Studies and Multimedia (Hamilton, Canada) and McGill University’s Centre for Interdisciplinary Research in Music, Media and Technology (Montréal, Canada). The signal from Ogborn’s live coding is monitored and calibrated for delay so it will match up with the signal arriving from Montréal. At both ends, the effect is one of playing along with another in sync [18].

JackTrip, the application which provides the audio link and helps to manage latency issues belongs to a class of applications called network music performance applications [19]. These applications allow a real-time audio communication over a computer network for musicians in different locations to perform as if they were in the same room [20]. Performers connect through high fidelity multichannel audio and video links as well as specialized collaborative software tools [21]. Although these applications were not built to replace traditional live stage performances they do provide a novel way of collaboration which could see more mainstream adoption within the field of live coding.

There are technical factors that must be considered while using network music performance applications. These factors include bandwidth demand, latency sensitivity and audio stream synchronization. High definition audio streaming is used by these applications to provide the most realistic sound possible. This comes at a cost however as high definition audio streaming is considered to be one of the most bandwidth demanding uses of today’s networks [22]. Latency is introduced when sending audio which is processed on a local system and then sent across a network. In order for a performance to feel natural to the human ear, latency must be kept under 30 milliseconds [23]. Synchronization cues are used by network music performance applications to account for long latency situations, such as delaying audio so that it appears to sync up better. Audio stream synchronization tries to synchronize multiple audio streams from separate locations at the endpoints to form a consistent representation of the music. This is the hardest technical issue today for these systems.

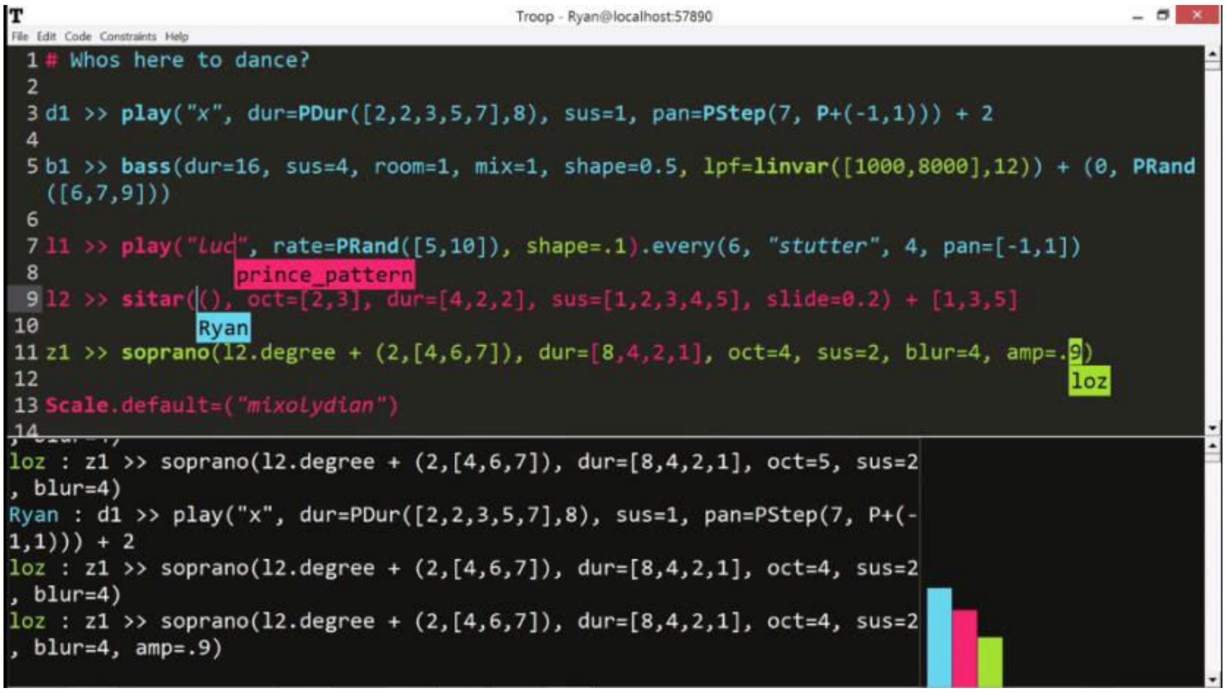
The need for tight synchronization when dealing live coding is not as strict as with traditional instruments. Live coding has a more improvisational nature to it with many long meandering sounds which blend into each other. This allows live coding to be slightly more flexible when it comes to issues of latency and synchronizing in collaborative environments as these issues are harder to detect for the human ear.

When performers wish to collaborate in the same room, such as in at a live concert, the issues of latency and synchronizing are not as big of a problem in this setup as performers can used the same local network. Performers may choose to create sounds separately in the same room using different live coding environments or simply just different text editors with the same environment each with their own laptop. Depending on the style of music this requires each laptop to be temporally synchronized in real time. A common method of doing this is through the use of a shared clock protocol where each machine constantly listens to a designated time-keeper to keep in sync. This method has seen implementation in the Birmingham Ensemble for Electroacoustic Research or BEER for short [13]. This requires an additional layer of configuration which is still liable to lag between performers depending on the bandwidth of the network being used.

This approach to collaboration has also seen implementation outside of the live coding sphere in performance ensembles called “Laptop Orchestras” [24]. While sharing the similar aspect of having collaborative music through computers, laptop orchestras don’t necessary use the traditional live coding environments to achieve this. Princeton University has helped to popularize laptop orchestras with the introduction of their first performance in 2006 [25]. The Princeton Laptop Orchestra, or PLOrk, takes the traditional model of an orchestra and reinvents it for the 21st century. Each performer uses a laptop and custom designed hemispherical speaker that emulates the way traditional orchestral instruments cast their sound in space. Connected through a local wireless network and directed by composer and instrument designer Jeff Snyder, and features new electronic instruments that arise from his research [26].

The instrumentation in PLOrk is highly experimental with performers using the motion control data information from Nintendo Wii remote controllers to play instruments or the tracking of a mouse cursor to create sound. PLOrk takes a somewhat collaborative control approach to collaboration as performers interact with an interface as opposed to directly editing code.

This type of multiple user ensembles where performers create sound independent of each other locally on their own system while using some sort of shared clock or server to keep in sync has been criticized by some in the live coding field including the founder of Troop [8]. Ryan Kirkbride, the founder of Troop, argues that as well as still being liable to lag, these systems can lead to poor audience experience. Multiple live coders working on separate screens may result in a non-optimal audience experience because too much is going on or there isn’t enough projector space for all screens to be displayed. Ryan Kirkbride has concluded that the best way for live coding collaboration to be performed is with a shared text buffer such as with Extramuros [27] or Troop as this system would allow for all code to be displayed together in an easy to read and understand format for the audience. As shown with Troop, see Figure 1.



```
# Whos here to dance?

d1 >> play("x", dur=PDur([2,2,3,5,7],8), sus=1, pan=PStep(7, P+(-1,1))) + 2

b1 >> bass(dur=16, sus=4, room=1, mix=1, shape=0.5, lpf=linvar([1000,8000],12)) + (0, PRand
([6,7,9]))

l1 >> play("Luc", rate=PRand([5,10]), shape=.1).every(6, "stutter", 4, pan=[-1,1])

l2 >> sitar((), oct=[2,3], dur=[4,2,2], sus=[1,2,3,4,5], slide=0.2) + [1,3,5]

z1 >> soprano(l2.degree + (2,[4,6,7]), dur=[8,4,2,1], oct=4, sus=2, blur=4, amp=.9)

Scale.default=("mixolydian")
```

```
loz : z1 >> soprano(l2.degree + (2,[4,6,7]), dur=[8,4,2,1], oct=5, sus=2
, blur=4)
Ryan : d1 >> play("x", dur=PDur([2,2,3,5,7],8), sus=1, pan=PStep(7, P+(-
1,1))) + 2
loz : z1 >> soprano(l2.degree + (2,[4,6,7]), dur=[8,4,2,1], oct=4, sus=2
, blur=4)
loz : z1 >> soprano(l2.degree + (2,[4,6,7]), dur=[8,4,2,1], oct=4, sus=2
, blur=4, amp=.9)
```

Figure 1 [14]

Troop in action, three users collaborate indicated using different coloured code for which user.

2.4 Collaboration Conclusion

There is still a lack of consensus on how collaboration within live coding should be done. As we have seen, this area is still going through a development stage as different methods such as a shared clock, a shared text buffer and collaborative control compete to become the standard approach to collaboration in live coding. While the field of collaboration within live coding is still developing and growing, the technologies that surround and supporting it such as bandwidth speeds, networked music applications and live coding environments grow too. Designs and approaches will continue to ebb and flow as a universal standard is forged over time.

2.5 Supercollider

Supercollider is an environment and programming language for real-time audio synthesis and algorithmic composition originally development by James McCartney in 1996 [28]. It is a free and open-source software that is at the bedrock of many live coding environments today such as Sonic Pi, Tidal Cycles and ixi lang allowing them to modify and execute code during runtime as well as produce sounds [2]. The Supercollider environment is split into two components: a server, scsynth and a client, sclang. These components communicate using OSC (Open Sound Control) § 3.2.3 OSC for more on OSC.

All the sounds you hear in Sonic Pi are produced by the Supercollider synthesis engine. Sonic Pi's server, written in Ruby, keeps track of all the details of active notes, synthesizer parameters, FX, samples, and so on, and feeds all that data to Supercollider via OSC protocols. Supercollider is almost untouched by Sonic Pi developers apart from one patch to the network code to keep it from aggravating the Windows Firewall [29].

Chapter 3

Design

3.1 Design Overview

Due to the nature of this project it was necessary to first take an exploratory and experimental phase to get an understanding of the technology stack as well as answers to some important questions. What would be possible? What has been done? What can be done? What software would be useful? What is the complexity of this project?

Unlike in web development for example, there was no standardized design procedure and a lack of community surrounding the idea of the project. As such the project first had to be broken down into a set of problems and initial choices had to be made. Firstly, which live coding environment I would be developing in. Secondly, what are the methods of collaboration that exist already and the scope of methods that are possible? Thirdly, the ways in which audio information can and has been transmitted, and lastly, assuming the use of a server to handle multi-user interact, what language should that server would be written in? With these initial questions I began my research evaluating the complexity of tools and methods of collaboration. The exploration and experimental phase lasted from September 2017 to mid-January of 2018.

The second phase, Design Architecture, takes all the research and information from the exploration phase and applies it to the project to create a collaborative environment for live music composition

3.2 Exploration and Experimentation

3.2.1 Tidal Cycles

Initial research and experimentation began with the live coding environment Tidal Cycles [30] in the beginning of the Michaelmas Term of 2017. It became clear early on that Tidal Cycles was going to be a difficult piece of software to get working let alone develop in. While popular in the live coding music field, its overly complex setup and runtime execution eventually led me to rule it out after several weeks of troubleshooting. The instruction manual on the website is out of date and offers suboptimal installation practices such as telling the user to use Haskell and edit the cabal config file instead of using the widely popular and preferred software Stack which manages dependency issues for the user. The instructions also make suggestions which are out of date recommending incorrect versions of Atom and Tidal Cycles. Six different applications (Haskell, Atom, Supercollider, SuperDirt, Git and Tidal Cycles) need to be installed in order to run Tidal Cycles and they all have to be within certain acceptable versions which caused massive issues when using windows or Ubuntu as my operating system.

Ubuntu Tidal Cycles with all its dependencies would only run on the unstable 17.04 version of Ubuntu. Issues with how the audio system interacts with the hardware, listed as a problem with jackd2, caused so many problems and coupled with the lack of support on troubleshooting this issue led me to move platform to Windows. On Windows Atom and Tidal Cycles had issues with version types. It was unclear if Tidal Cycles or Atom was a version ahead but the result meant that it didn't run code without hacking the code and overwriting the way the Tidal Cycles packages in Atom ran the code. After multiple weeks of installing, reinstalling, updating Ubuntu, switch from Ubuntu to Windows 7 and reinstalling everything, I finally got Tidal Cycles to make a sound. At this point the interaction between Atom and Tidal Cycles broke down causing Tidal Cycles to only work for solo sound operations because of the way I had hacked the code.

At this stage of the project, Tidal Cycles had demonstrated enough issues to warrant scrapping it as the environment of choice for development. I began looking at alternative live coding software and that is when I found Sonic Pi. I installed an exe file from the website, ran the installer, launched Sonic Pi and could instantly begin making sounds.

While Tidal Cycles may have its advantages, such as having extreme shorthand for all its commands and a powerful way of creating musical patterns, it's out of date instruction guide coupled with its very complex set up and running operations forced me to rule out the application as my choice for development.

Many live coding environments are overly complex in their setup and execution leading to their unpopularity with the mainstream audience. I think it is Sonic Pi's simplicity in both setup and execution that has enabled it to be more widely adopted and used. Sam Aaron addresses this in a talk he did at OSCON 2015 in Amsterdam [31], where he talked about the "low barrier to entry" of Sonic Pi due to it being a free open source application with a simple "easy to get started" setup and execution.

Software	Installation Complexity	Number of Components	Execution Complexity	Language Syntax Complexity
Tidal Cycles	High	6	High	Mid
Sonic Pi	Low	1	Low	Low

Figure 2

Comparing Tidal Cycles and Sonic Pi

3.2.2 Collaborative Editing vs. Collaborative Control

As well as this project originally beginning with Tidal Cycles, the original method of collaboration at the start of the project was to be collaborative editing. Drawing very close parallels to Troop and Extramuros, performers would use a shared text buffer allowing them to edit each other's code in real time. This approach was discarded after research due to the following reasons:

1. Complexity, the implementation of such a system would be outside the scope of a Final Year Project. This was made apparent after researching this method and seeing that a PhD student from the University of Leeds was currently developing such a system which he has named "Troop" for his PhD. Showing that this is a deep and rich problem.

2. Limits the audience because collaborative editing requires a deeper knowledge of Sonic Pi and its features. This can create barriers to entry for mainstream users who are unsure about how their edits affects another users' code.

3. It's already been done. Applications like Troop and Extramuros are already in development and provide this solution to the problem of collaborative composition.

Collaborative control on the other hand, would solve these issues of audience limitation, lack of novelty and the out of scope complexity. In a collaborative control environment one user would have access to the code while all other users would use some sort of interface to indirectly interact with the code to produce sound. This led to a new challenge of needing to design and develop an interface as well as figuring out how it would communicate.

3.2.3 OSC

As discussed in Chapter 2.2, Open Sound Control (OSC) has become a de-facto standard for musical communication. It is a digital media content format for streams of real-time audio control messages [11]. OSC has also found uses in show controls and robotics.

The format structure of an OSC message is broken down into three stages. An OSC message, which contains an address e.g. the location of a button, and data which is a common 32-bit binary encoding for integers, real numbers and text. An OSC Bundle, which houses a number of messages, each of which represent the state of a sub-stream at an enclosed point in time called a timetag. Finally, an OSC stream, which are sequences of bundles defined with a timetag. (Figure 2)

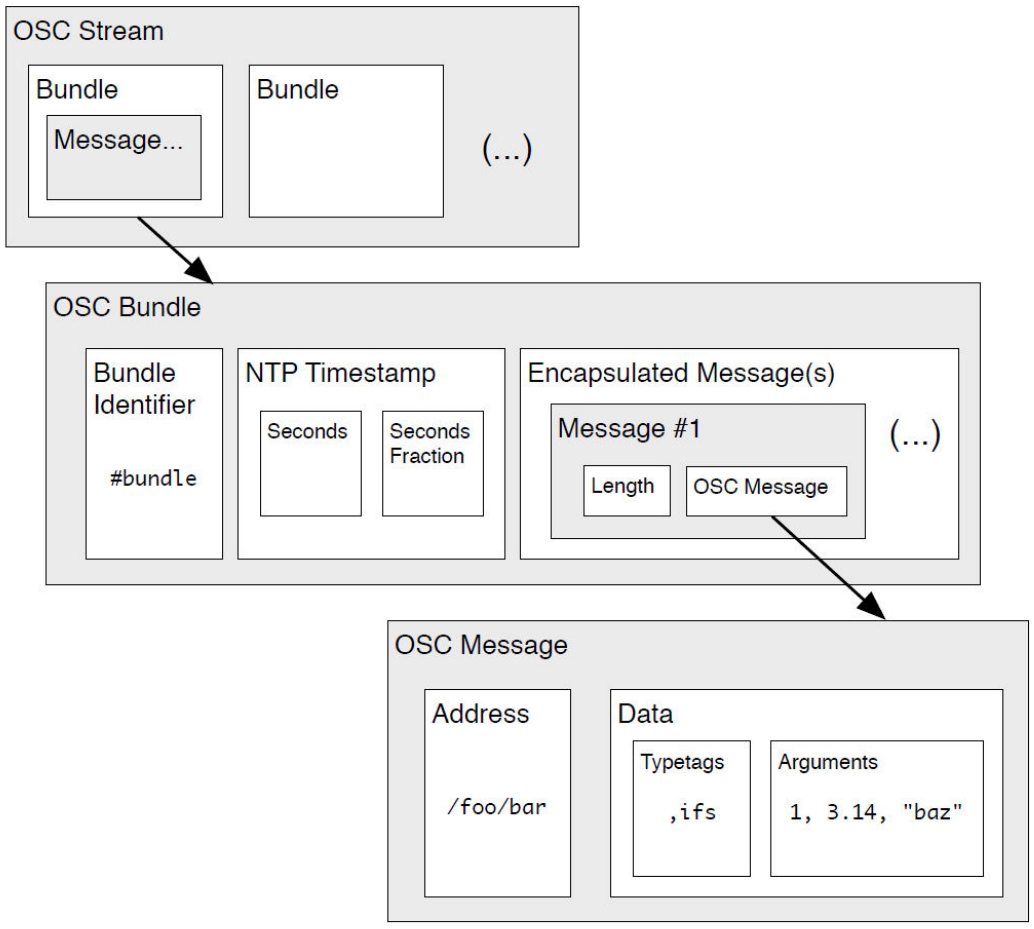


Figure 2 [11]

Overview of OSC protocol for sending messages

Due to the time sensitive nature of the information being transported OSC uses UDP/IP to transport information. UDP is faster than TCP but at the cost of being less reliable due to UDP not using acknowledge packets (ACK) to permit a continuous pack stream. Each packet is delivered in its entirety or not delivered at all. With live sound, milliseconds count and it doesn't take a skilled ear to hear when things are even a little later than they should be. Therefore, in the case of live coding it is better to be missing a sound then have it appear late.

To fix the case where packets may arrive out of order, OSC bundle timestamps can be used to recover the correct order. If the packet exceeds the maximum transmission unit (MTU) the packet may be fragmented over multiple pieces. "This fragmentation can introduce extra delay as UDP/IP must reassemble the pieces before delivering the packet to an application." [11]

3.2.4 Independent Synchronizing

Irrespective of the collaborative method a server had to be set up to manage step data and pass information to Sonic Pi for playback as well as being able to keep all connected devices up to date. From researching this topic it became clear that having the synchronization happen independently to the live coding environment was generally the preferred method [5] [4]. This is due to the many unique ways live coding environments can operate. If you code the collaboration and synchronization for one environment, it may not be useable in another environment. By having synchronization done separately on a server this project can be reworked to support any live coding environment that supports the OSC protocol. Furthermore, with no development document on how Sonic Pi is wired together, attempting to build the collaborative environment into Sonic Pi would have proven to be very complex if not impossible due to the very limited documentation on it.

I decided on Node.js as my language of choice for development of the server for the following reasons. One of the main advantages of Node.js is that it supports multithreading. The asynchronous and event-based architecture of Node.js made it a great fit for RTA (Real-Time Applications) as was the case for this project with real time audio messages being used to communicate between all parts of the system. The architecture of Node.js is built to handle requests which are happening concurrently and where data is frequently shuffled back and forth from the server to the clients. This was exactly what was needed for the design of this system as devices and Sonic Pi would constantly be sending and receiving messages to the server.

The challenge of working in Node.js was the steeper learning curve when compared to other languages such as PHP, and the different approaches to programming problems such as the use of referencing instead of copying of methods to increase performance and memory. I spent two weeks just coming to grips with how Node.js operates because of this, but I believe it was the best and most suitable choice for this project in the end.

3.3 Design Architecture

The second phase, Design Architecture, takes all the research and information from the exploration phase and applies it to the project to create a collaborative environment for live music composition.

3.3.1 Sonic Pi

With Tidal Cycles ruled out and having researched a number of alternative live coding environments I finally settle on Sonic Pi. As previously stated in §3.2.1 Tidal Cycles, Sonic Pi’s development friendly, simple installation and popularity within the field lent itself to be the obvious choice after abandoning Tidal Cycles.

Although the creator Sam Aaron has stated that there is currently no official development document describing the architecture of Sonic Pi [32]. Users within the community have created a rough diagram of the architecture of Sonic Pi which Sam Aaron has approved (See Figure 3), although he warned that due the evolution of the code over the lifetime of its development the architecture and abstractions are not in the easiest form to understand.

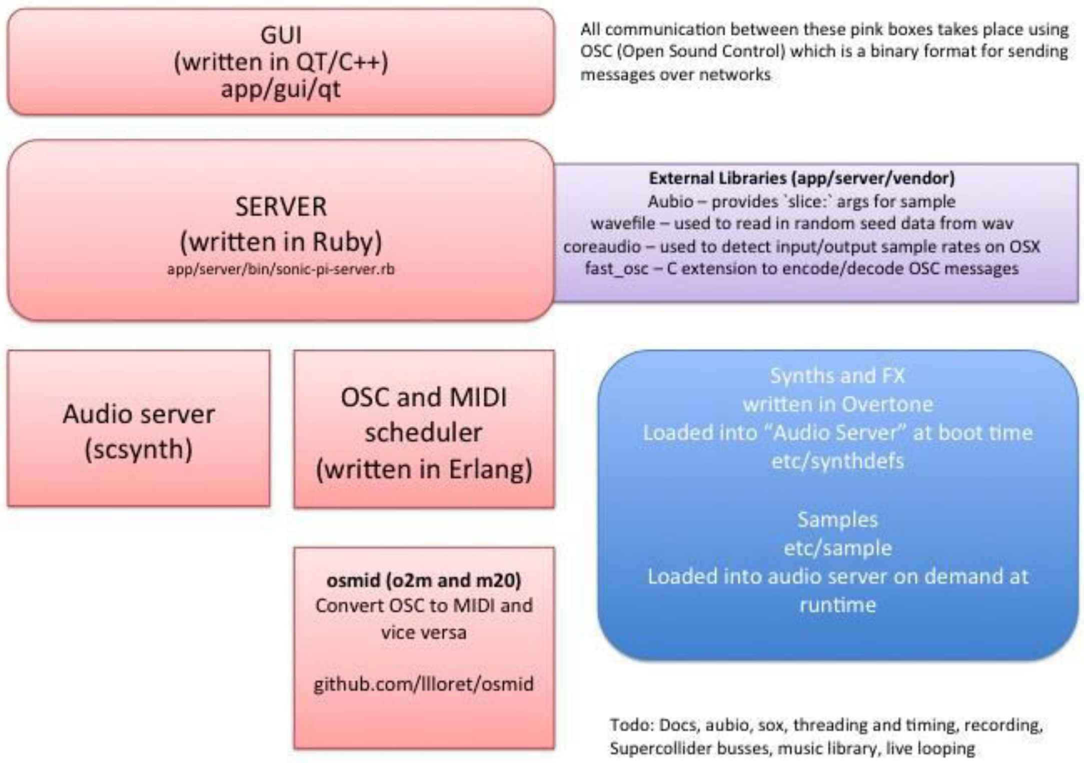
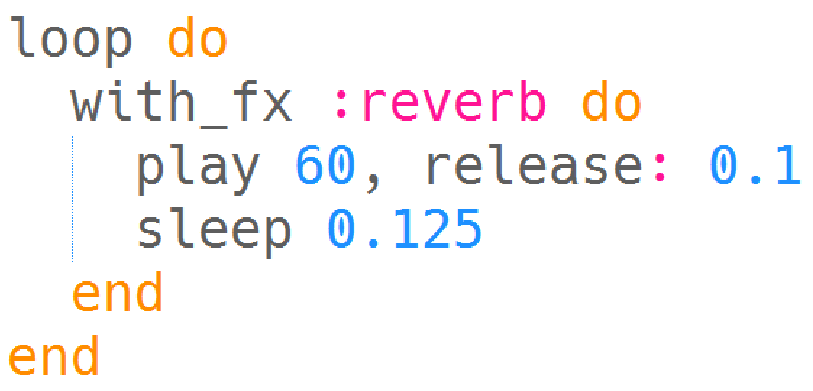

GUI
(written in QT/C++)
app/gui/qt
All communication between these pink boxes takes place using OSC (Open Sound Control) which is a binary format for sending messages over networks
SERVER
(written in Ruby)
app/server/bin/sonic-pi-server.rb
External Libraries (app/server/vendor)
Aubio – provides `slice:` args for sample
wavefile – used to read in random seed data from wav
coreaudio – used to detect input/output sample rates on OSX
fast_osc – C extension to encode/decode OSC messages
Audio server
(scsynth)
OSC and MIDI
scheduler
(written in Erlang)
Synths and FX
written in Overtone
Loaded into "Audio Server" at boot time
etc/synthdefs
Samples
etc/sample
Loaded into audio server on demand at runtime
osmid (o2m and m20)
Convert OSC to MIDI and vice versa
github.com/lllloret/osmid
Todo: Docs, aubio, sox, threading and timing, recording, Supercollider busses, music library, live looping

Figure 3 [32]
Overview of Sonic Pi Architecture

The internal architecture of Sonic Pi has a lot of complexity with multiple different languages being stitched together. This made me feel like I didn't want to mess with the underlying architecture unless absolutely necessary. Especially because there is no development document to give detailed explanation of all parts of the system and how they are wired together.

The architecture of Sonic Pi has resulted in a few limitations, such as a undefined line limit and high CPU power consumption. The line limit issues arise when a program exceed somewhere between 490 to 530 lines of code. The error results in the program not running until the user drops the total lines used to below an arbitrary value somewhere below 530 lines depending on the operating system. This issue is known to the developers and is caused due to the architecture of Sonic Pi. Sonic Pi uses UDP to communicate between the GUI, the server and scsynth. There is a size limit on the UDP packet which can be sent between here. Work has gone on in earlier versions of Sonic Pi to explore the use of TCP for communications instead of UDP to overcome this issue. However, because of the handshaking involved with TCP it is inherently slower, and when dealing with music a delay can prove to be detrimental to a performance. The developers are still not yet happy with the performance when using TCP and this issue remains on the to do list for the developers. This later became an actual issue during implementation which needed to be addressed, see §4.5 Design and Sonic Pi.

The other major limitation caused by the architecture to developing in Sonic Pi is CPU power consumption. This is caused for two reasons, too many threads running concurrently and the use of lots of FX effects in a program [33]. Of these two reasons the most reliably way to run out of CPU power is to use lots and lots of FX. With every with_fx that is called, a new FX synth is triggered which as its own lifespan, together these FX effects can quickly drain a CPU. Consider the following code:



```
loop do
  with_fx :reverb do
    play 60, release: 0.1
    sleep 0.125
  end
end
```

In this code I am playing note 60 with a very short release time. I want reverb so I have wrapped it in a reverb block (with fx :reverb do). When the code runs, firstly I have a loop which means everything inside of it is repeated forever. Next, I have the with_fx block. This means I will create a new reverb FX every time I loop. This is like having a separate FX reverb pedal for every time you pluck a string on a guitar. All the work of creating the reverb and then waiting until it needs to be stopped and removing is all handled by with_fx for the user, but this takes up CPU power, especially when you have multiple FX effects running concurrently within the same loop.

Despite the criticism of Sonic Pi's limitations it is actually a powerful live coding environment. Sonic Pi is a multithreaded system, allowing users to schedule multiple threads and live loops for creating looping threads which can be edited at runtime. As well as keeping threads in the same tempo, it is important to keep them in sync or phase with each other. Sonic Pi can carry out both of these actions on behalf of the user. This idea of phase isn't a typical requirement of programming languages, so Sonic Pi had to invent a few concepts: cue and sync [34]. Sam Aaron, the founder of Sonic Pi, describes thinking of these two concepts as a conductor and an orchestra. The violins wait for their cue to start playing and only begin when the conductor indicates to them do so. In other words, the violins are synchronizing on the conductor's cue [35]. This is a powerful tool and takes a lot of the messiness of synchronizing threads in Sonic Pi away from the user.

Another way to see the power of Sonic Pi is to see it in action, a demonstration of this is available on YouTube, where a user, Sebastien Rannou, played the entire of Aerodynamic by Daft Punk live using Sonic Pi [36]. Sam Aaron also has many recordings of himself performing live music performances to audiences using Sonic Pi [37] [38].

3.3.2 Collaborative Control

Having ruled out Collaborative Editing and instead deciding to settle on a Collaborative Control System, I now had to design a system for this method of collaboration to work in. It would have to allow one user to have direct control over Sonic Pi with the ability to change and alter the instrument and sample sounds in real time through code, while allowing all other users to create and manipulate sounds by playing notes and using effects such as attack, release, pitch and volume through an interface. I decided I would try having the interface work on mobile devices after coming across an application called TouchOSC which uses OSC to send messages.

This approach would require only one user to be knowledgeable in how Sonic Pi works while all other users could simply connect with their phones and begin making music through a display interface, with no requirement to understand how Sonic Pi works.

3.3.3 TouchOSC

With the decision made to use a collaborative control system and having explored the use of OSC in my project I came across TouchOSC. TouchOSC is an mobile application which allows the user to send OSC protocol messages from their mobile device. The UI design on the application is high configurable with a desktop editor to allow the user to create UIs and push them to the application. I designed my UI to be intuitive and user friendly having tested early versions of the design on my fellow students to receive feedback and criticism as well as to catch bugs in the UI such as push buttons causing large delay between devices and Sonic Pi. The final version allows user to play with three different synths, drums and a range of effects which can be applied to both the synths and the drums, see §4.5 Design and Sonic Pi.

3.3.4 Distributed State

The design of my system had to deal with distributed states and the issues that can arise when dealing with them. Issues such as packets loss, latency and different states falling out of sync needed to be addressed before implementation could begin.

The issue of packets not arriving was exacerbated by the fact that communication had to be done over UDP due to the nature of the system dealing with music. If the system was to use TCP, it would create too much latency due to the handshake that is required with TCP. This meant the packets could be lost with no method of recovery in place for the lost packet. However for my system, packet loss is predicted to be relatively low because all aspects of my system are on the same local network. The design of the system prioritizes low latency over consistency because of this reason.

At no exact time is there any promise of consistency. However, through the process of eventual consistency, as long as a user can keep processing messages the system will settle down and become synchronized again. Furthermore, if a user, for example, changes a volume slider from a low setting, say 0.2000 to a high setting, 0.8000, there is over 50 messages sent to Sonic Pi as well as all other connected devices from the device conducting the change of the slider value as it moves from 0.2 up to 0.8. This means that even if another device drops a packet or two at the very end it will still be very close to the correct value of the volume slider. If a device drops a packet in the middle of the change it will be corrected by the continued flood of incoming packets.

When dealing with a toggle button however, there isn't as much leniency. A toggle button only sends one messages for on and off, 1 and 0. If another device misses one of these packets they will be completely out of sync. There are some safety precautions for these situations and whatever Sonic Pi is playing will always be the true state of the system at any point in time. In the case where a user turns a toggle button on but the packet is lost on the way to Sonic Pi then the user can just turn the button off and on again with no error to get Sonic Pi to play the note. In the case where user A turns on a toggle button and Sonic Pi receives the packet and produces the required sound but the packet is lost on the way to update user B of this change made by user A, user B is then out of sync with the system. Recovery of synchronization of user B with the system is dealt with by the use of eventual consistency. However, to avoid the issue of duplication caused by out of sync devices, if user B then turns on the same toggle button as user A has already turned on, Sonic Pi will check and observe that that thread is already running and perform no action. They are now back in sync with each other. Further methods to improve consistency across the system are addressed in §5.2 limitations.

3.3.5 Finalized Design

The finalized design (figure 4) combines all the technologies discussed above to produce a collaborative control environment for Sonic Pi. The server takes information from all connected devices as well as Sonic Pi and manages the step data. All information from mobile devices are sent to Sonic Pi for playback as well as to all other connected devices to keep them all synchronized with each other. Sonic Pi was programmed to allow it to send information to all connected devices to allow for resets of all effect and button between runs however a bug I discovered within the Windows version of Sonic Pi prevented this from happening, see §4.5 Data Sharing between Devices

The connection of all parts of the system is done through using a local IP address with different port numbers. Devices connect and send information to the server with port 8000 and receive information from the server through port 9000. The Server connects to Sonic Pi on port 4559 and receives information from Sonic Pi through port 7000.

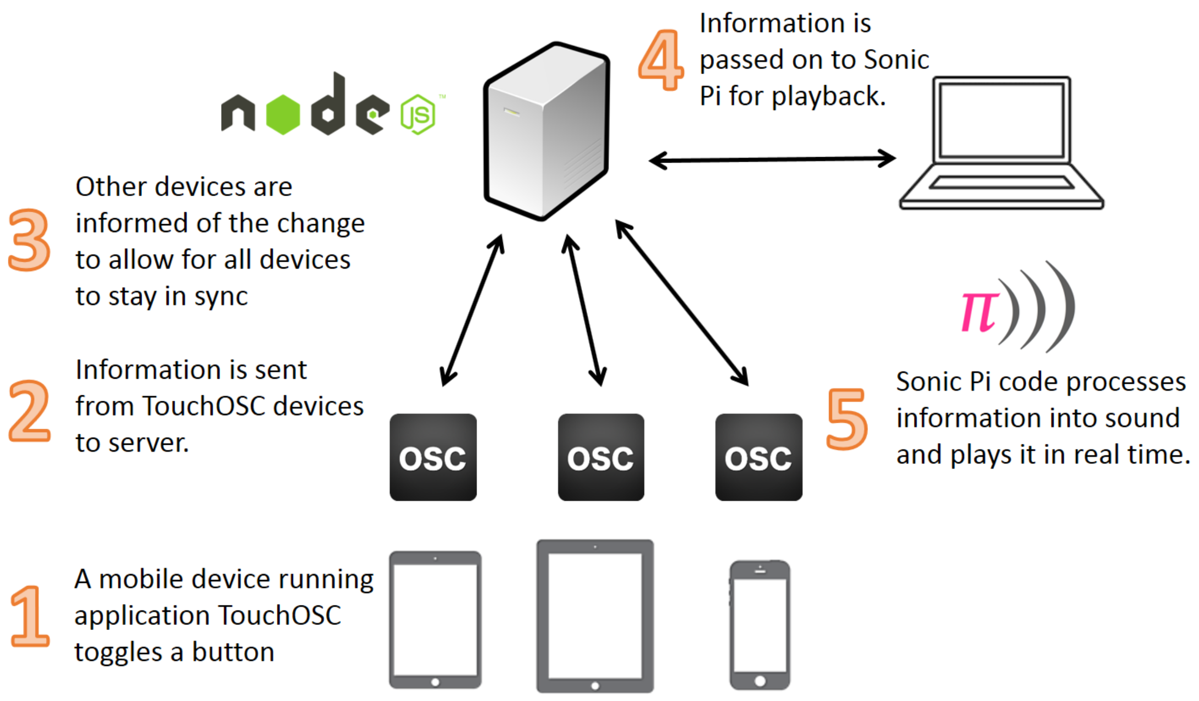

node JS
4
Information is passed on to Sonic Pi for playback.
3
Other devices are informed of the change to allow for all devices to stay in sync
π)))
2
Information is sent from TouchOSC devices to server.
OSC
OSC
OSC
5
Sonic Pi code processes information into sound and plays it in real time.
1
A mobile device running application TouchOSC toggles a button

Figure 4

Design Overview of my Collaborative Control System

Chapter 4

Implementation

4.1 Implementation Overview

From the beginning of the development I broke down the implementation of this project into milestones with target dates of completion for each aspect. The following sections in this chapter examine each of these milestones discussing the issues and solutions that I uncovered.

4.2 TouchOSC Connecting to Sonic Pi

With the research and design finalized, I began development taking the following steps:

1. Download the TouchOSC desktop editor and create a basic layout to use during the development and testing portion of my project.

2. Push the created layout to my mobile device and then code Sonic Pi to produce output sounds when specific OSC messages were received from the mobile device from button presses.

3. Connect my mobile device to Sonic Pi directly via the application TouchOSC by setting the correct port numbers in TouchOSC to allow for the messages to be sent to Sonic Pi.

This proved to be relatively straightforward due to a recent update of Sonic Pi which allowed it to now support external OSC messages through the use of port numbers from an external device such as a MIDI. However due to this being a new feature of Sonic Pi there were very little resources on the correct way to implement this.

4.3 Connecting Multiple Devices to a Node.js Server

With Step 1 complete, Sonic Pi making sound from the controls of a mobile device, the next step was to begin development of the server to allow for the connection of multiple devices. This was the most difficult and largest part of the development effort in the project as I had no real idea how to implement this and the possibilities and limitations of such a system.

I began by getting a local HTTP server running to gain an understand of how node.js works, using methods and functions to become more familiar with the language. With this basis I began working on creating a server to allow for the connection of multiple devices. One of the big breaks was discovering osc-js, which is an Open Sound Control library for JavaScript application with address pattern matching and timetag handling. It allows messages to be sent via UDP and WebSocket or both. This library provided the foundation for the development of the server.

With a foundation to build from and with numerous attempts made in trying to connect my mobile device to the sever, I managed to set up a node.js server which uses UDP listening on port 8000 for devices transmitting OSC messages, done via TouchOSC, with their outgoing ports also set to 8000. The terminal displays when server is up and when a new device connects the server prints the device's IP address. Since each device has its own unique IP address I just check that the current connecting device isn't already connect to the server and if it isn't, I connect it. This was the first big hurdle to overcome and the next challenge I had, was to take actually input from the devices to the server. Taking input from devices to the server proved to be less challenging then I first thought. Node.js was actually already taking in all the input from devices in real time I just didn't realize this. I created a variable called data which takes the information from osc.fromBuffer(msg) which is the value being sent to the server from any device. I then had this print out to console. Next step was to have the node.js server tell Sonic Pi what button has been pressed and then have Sonic Pi play a sound based off the button pressed.

4.4 Connecting Server to Sonic Pi

Although I now had devices all connected and sending information to my server I didn’t quite understand the format of how Sonic Pi reads messages as well as how to send messages from a server to Sonic Pi.

The first attempt to send information to Sonic Pi from the server using local host as the IP address and 4559 as the port number for Sonic Pi inside the parameters for the sock.send function did not work. It turned out Sonic Pi has a specific IP address similar to way all devices have which could be gotten through the Sonic Pi application. Having changed the IP address from local host to the specific Sonic Pi IP address I could send messages to Sonic Pi. However, the messages being sent could not be executed on by Sonic Pi. The value of the button press from a device wasn’t being read.

The initial idea of sending the message to Sonic Pi consisted of sending the address as a string and the value as a float but this wasn’t working. The sock.send function doesn’t allow for the sending of float values, it only allows string, buffer and unit8array. So I instead tried to send the address and value as a string but this caused Sonic Pi to read the message as two separate messages, one with an address and no value and one with no address and a value. This sent me back to the drawing board to reexamine the TouchOSC messages and look closer at how messages were being sent from TouchOSC to the server to understand how OSC messages are sent in general. I discovered that OSC messages are sent inside a type called Buffer. With this knowledge I went back to the server and packaged the address and float value of the messages into a var called msg. Then I converted msg into type buffer and sent that with the port number and IP address of Sonic Pi. This worked and Sonic Pi was able to provide playback for messages coming from the devices via the node.js server.

Having established the ability to send messages from multiple devices to Sonic Pi via a node.js server a new issue arose. Devices weren't being kept up to date on what other connected devices were doing which was leading to errors of message duplication and contradictions of the state of a button between devices. The solution to this problem was to implement data sharing between devices.

4.5 Data Sharing between Devices

As stated, data sharing between devices needed to be implemented to overcome to duplication and conflicting button state issues. Originally I thought I could do this through Sonic Pi, where Sonic Pi would send the information of one device button press to all other connected devices via Node.js. This approach proved to be overly complex and not possible due to a bug I discovered within the Windows version of Sonic Pi. On the windows version, Sonic Pi cannot send OSC messages only receive them. This is an issue I flagged on the Sonic Pi community forum, in_thread [39] and received a reply from the founder and creator of Sonic Pi, Sam Aaron, acknowledging that this is an issue on their systems too and seemed to only affect Windows users. There is currently no solution offered but they plan to have it fixed for the next release.

This discovery about the limitations of the Windows version of Sonic Pi prompted a new, simpler approach to the issue of data sharing. Instead of going through Sonic Pi I could do all the work through the node.js server and not even consult Sonic Pi. When a device is sending new information to the server to send to Sonic Pi I also have the server send the information to all other connected devices through port 9000 and using their individually stored IP addresses. I did this by having a device IP array list, which is a list of all connected devices' IP addresses. When sending information to Sonic Pi the server also cycles through the array list and sends the information to all other connected devices. When a new device connects to the server the array list is updated to reflect that change. This allows all devices to data share and keep synchronized with each other.

4.6 UI Design and Sonic Pi

With the framework for the project completely I began to focus my attention on the UI designs for the mobile devices as well as coding for Sonic Pi. This portion of the project helped to highlight limitations of Sonic Pi such as Sonic Pi's line limit, the heavy CPU power required and the inability to end live loops as well as helping me gain a deeper understanding of the language of Sonic Pi. The UI design went through many revisions throughout the development of this project as I gained feedback and criticism from friends and fellow students.

Early designs focused on creating realistic instrumentation, using a miniature piano keyboard with 13 keys, C to C, for both a piano and a bass piano (see Figure 5). This proved to be a mistake for a few reasons:

1. Latency: Trying to play the piano in real time using push buttons produced too much noticeable latency, with the delay between pressing a key and hearing it being very off putting for the user.

2. Size: In order to have each key big enough so that a user wouldn't mispress a button I couldn't add anything else to the screen such as volume and pitch effects. This limited the features and experimental ways in which sound could be produced.

3. Usability: Even for users that had 10+ years playing piano they found the controls of the mobile piano to be difficult to play due to the lack of action on the keys as well as the keys being too small and limited in range. Almost all piano users complained about being limited to only one scale.

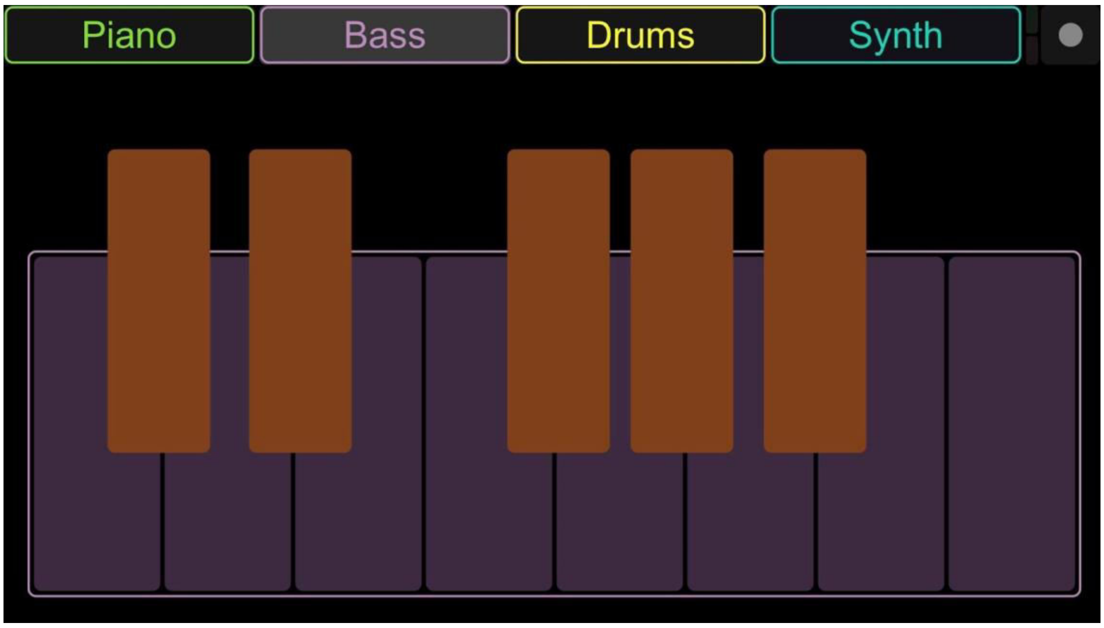


Figure 5

Bass Piano from first UI design

Equipped with meaningful feedback from my first UI design I decided to go back and do a complete redesign of the UI taking all that I had learnt to date into consideration. The new UI designs focuses on the use of synths with toggle buttons and effects to help expand the range of sounds that can be made. I also added a pitch shifter to allow for a larger range of notes to be played (See Figure 6 to 9). Using toggle buttons helped to decrease the latency to an acceptable level and made it easier for users to select the correct button.

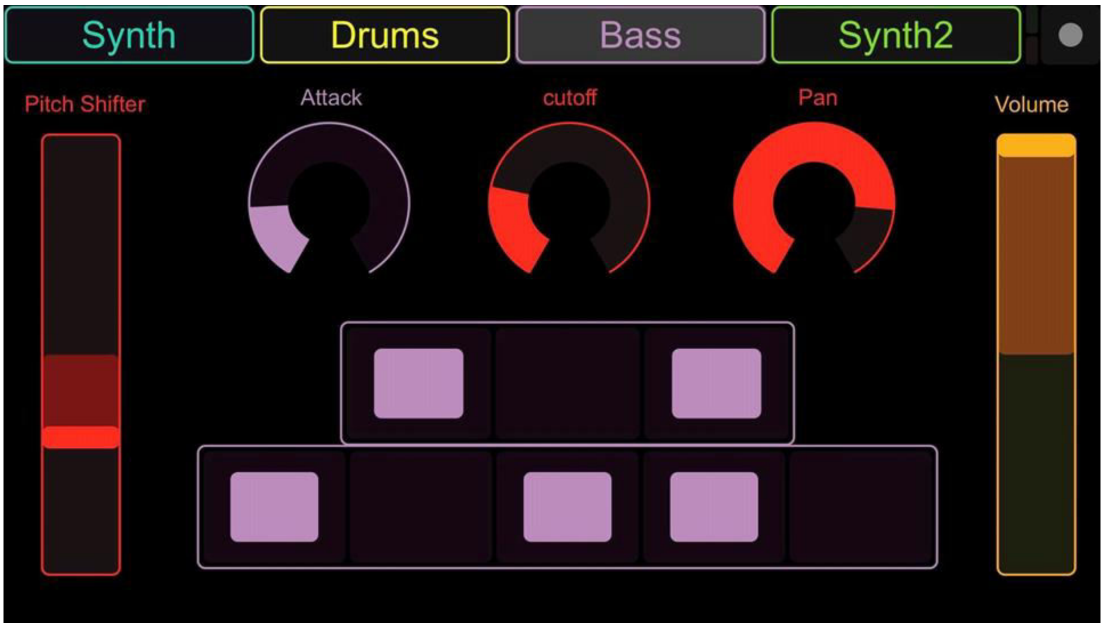


Figure 6

Bass from final UI design

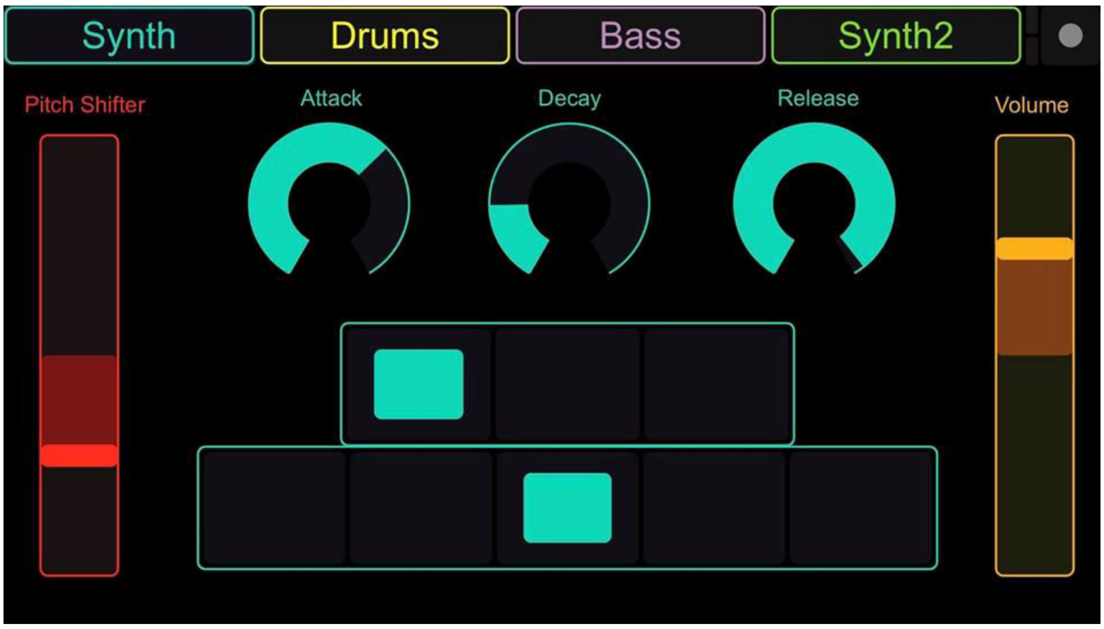


Figure 7

Synth from final UI Design

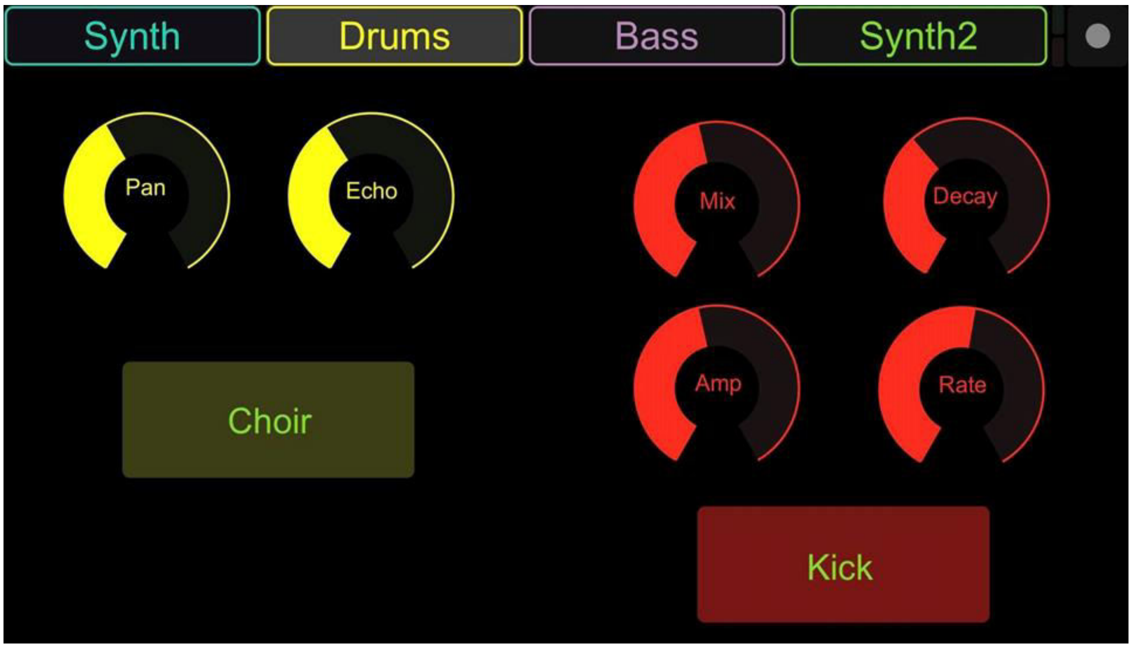


Figure 8

Drums from Final UI Design

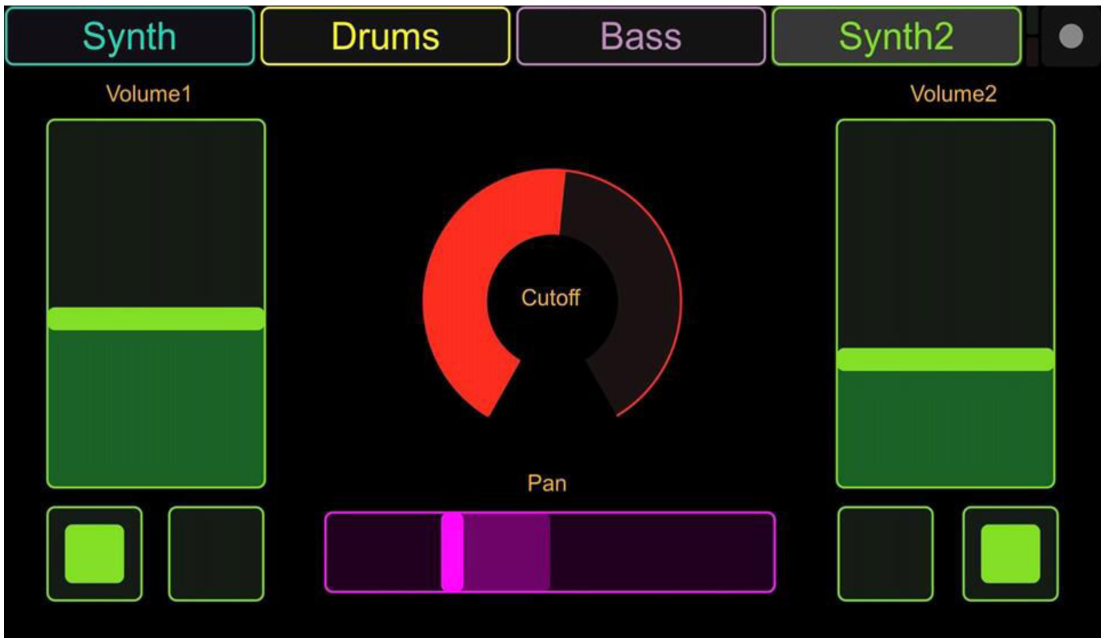


Figure 9

Synth2 from Final UI Design

However, this new UI implementation brought with it a variety of new problems to deal with when attempting code Sonic Pi such as CPU power issues, slider and effect issues, live loop errors and Sonic Pi line limit.

As mentioned in §3.3.1, an increase in the use of FX effects can cause a heavy load on the CPU. This new redesign took advantage of the numerous ways notes can be manipulated by FX effects and as a result development needed to be switched to a more power machine to allow for all these effects to be processed in real time. Also, as mentioned in §3.3.1, the issue of Sonic Pi's line limit now arose as a problem because of all the extra Sonic Pi code that needed to be written to map the new design. I found a number of ways to work around this line limit to get my code to run again:

1. Defining a function globally in one buffer so it is available to all of the buffers.

2. Arrange the code across two or more buffers and then use a cue command at the end of the first buffer to sync the start of the second.

3. Maximize the use of functions wherever possible throughout code in order to cut down on duplicate or wasteful lines.

Due to the use of sliders and effects throughout my system I ran into an error with Sonic Pi. Although on the devices the preset value for sliders and effects was zero, Sonic Pi was initializing them as NULL when the application started running. The reason for this initialization by Sonic Pi was unclear and this issue persisted for some weeks as I implemented workarounds until I realised that I had to initialize all values of the sliders and effects to zero at the top of the code in Sonic Pi as well as on the UI of all devices.

A live_loop in Sonic Pi is simply a loop that runs forever until deleted or killed using a kill function. Stopping a live_loop via an OSC command proved to be one of the most difficult parts of coding Sonic Pi. This is due to the fact that live_loops were not designed to be stopped via an OSC command and so in order to achieve it I had to think creatively. Normally loops are stopped by typing in a kill function at the bottom of the loop. This wouldn’t work for users operating via an mobile device as there is no option to type code. In the conventional use of Sonic Pi, threads are either run simultaneously or they are scheduled to run after a certain time using the sleep function. However in my implementation all threads are constantly listening for a change in a button’s value to allow them to run. When a button is turned on the thread runs its live_loop allowing a note to be played, but when a button is turned off the thread won't stop the live_loop as only a kill function can allow that. This kill function needs to be passed the note to kill so I had to save the note once the thread begins to run and then pass that note to a kill function inside an else statement once the button is turned off. This was the only way to stop live_loops via an OSC message.

Chapter 5

Results and Evaluation

5.1 Results

1. The objective of this project has been achieved by the creation of a completed collaborative environment which allows two or more users to simultaneously and remotely collaborate on live music composition using mobile devices and Sonic Pi.

2. The project has produced a working interactive mobile interface for use on both IOS and Android.

3. The project allowed me to design and create a server which can process OSC messages from multiple sources in real time and transmit that information forward which may have potential uses in other areas outside of live coding.

4. The project also resulted in the coding of fully functional Sonic Pi program which can interpret all OSC messages from the device interfaces and translate it into sound.

5. Stress testing on Sonic Pi was conducted which may have valuable insight to the development of Sonic Pi's next major release for Windows. Due to the unusual way in which Sonic Pi was used throughout the course of this project, such as all threads being declared at the beginning of the programs runtime, the use of multiple external devices for communication and the large of amount of code written and running at any one time. Less common issues such as the line limit and the previously unknown issue of the Windows version of Sonic Pi being unable to send OSC messages to external devices were uncovered by me and reported to the developers.

6. An extensive assessment of the application TouchOSC and the live coding environment Tidal Cycles was also carried out. TouchOSC which is normally used on one device talking to one computer and acting as a MIDI was reworked to work in a new way as part of an ensemble of devices talking to the same computer. Tidal Cycles is discussed in depth see §3.2.1 Tidal Cycles, and compared to Sonic Pi under several features.

Building off the existing work in the field of live coding and live coding collaboration, I believe this project will help to contribute to the field by the showcasing of another way to express live coding. Taking on board advice from those that have tried collaboration before, methods such as independent synchronization and the importance of data sharing is used in this project because of their acceptance and success so far in the field. Studying different approaches to collaboration such as Troop's collaborative editing technique helped to highlight for me the depth of complexity of collaborative editing and pushed the approach of this project into the direction of collaborative control.

The research required to be done and outlined in Chapter 2 in the literature review of this project exposed to me the extent to which research is being conducted in the area of musical coding such as Birmingham University's

Ensemble for Electroacoustic Research or BEER for short and Princeton University's Laptop Orchestra, PLOrk. As well as the numerous live coding environments that exist such as, ChucK, Extempore,Impromptu, Overtone, Supercollider, ixi lang, Tidal Cycles and Sonic Pi. It was clear to me from my research was there is active and engaged interest in research in the field of live coding. The research done in this project on methods of global collaboration and the issues that must be tackled could help provide useful insight to other developers looking to expand collaboration from a local network to a global one.

5.2 Limitations

In the course of this project I encountered some limitations with the technology I was using as part of my tech stack as well as some design limitations.

As discussed in the §4.5 UI Design and Sonic Pi the limitations of Sonic Pi were (a) its line limit, (b) the bug where Sonic Pi could not relay messages back to devices, (c) Sonic Pi heavy CPU usage when using FX effects and (d) the fact that the 3.0 version of Sonic Pi, which supports external OSC commands, was still in beta on windows for this project all limited development.

The design of this project currently requires all parts of the system to be on the same local network in order to run and so limits its application. Expanding this to become a global system could be a final year project on its own see §5.3 Future Work.

At no exact time is there any promise of consistency. As discussed in §3.3.4 Distributed State, there is no promise of consistency and the system instead works of off the idea of eventual consistency where given enough time the system will settle down and become synchronized again. This works fine for the system in its current state where all aspects of the system are on the same local network resulting in a relatively low level of packet loss. If the system was to expand beyond sharing the same local network this issue would need to be readdressed see §5.3 Future Work.

5.3 Future Work

As mentioned in §5.2 Limitations an expansion of this project to become a global system could be a future final year project. This of course would bring up new challenges of its own as well as heightening existing ones. One of the biggest of these challenges would be the issue of latency which could potentially be addressed using Ogborn's system discuss in §2.3 Methods of Collaboration, where synchronization is done at the endpoints for each user. This would also require a redesign of some aspects of the system for example, if performers were in different rooms they would each need Sonic Pi running to hear playback.

Another major obstacle with expansion of this project beyond a local network would be inconsistency. The issue of packet loss and parts of the system becoming out of sync would need to be addressed and a redesign of parts of the system would be necessary. There are a few methods that could be introduced to overcome this issue:

1. Every time a user opens a new page within the mobile interface information from the server is sent to that device informing it of the current state of that page.

2. The server could send a "resynchronize packet" every few seconds to keep all devices update to date on the current state of the system.

Both of these methods would use TCP rather than UDP to send the information from the Server to the devices to guarantee no packets loss. These approaches would require the server to save the most recent activity of all buttons and constantly be updating its list. These approaches would also need to be tested out and tweaked to manage issues of latency that may occur due to the use of TCP and potential issues of overloading the server due to its new secondary use especially as the number of active users increases.

Implementation of this project in schools to support Sonic Pi's ongoing effort. As mentioned in the description of Sonic Pi in §1.1 Introduction, Sonic Pi was designed to support both computing and music lessons in schools. I believe there is no better way to encourage children to get involved in computing and music than to create an environment where they can work together. Coding, especially in the beginning can be a very isolating activity where everyone works alone trying to get their own program working. This lonesome task can prove to be demotivating and boring to first time coders, especially children. By implementing my collaborative system into schools children can start to work together to tackle issues and code the type of music they listen to . A group could be broken into different sections with some children writing code in Sonic Pi while others create UI designs for mobile devices and some could even manage or keep track of the server, allowing them to see the different jobs that exist within the field of computing. By including multiple children under the guidance of one project goal, to code music, the task goes from being an isolating activity to a shared group experience. This helps children not only become interested in coding and music, as is the goal of Sonic Pi, but also learning about the importance of teamwork. It is standard practice in businesses and companies for coders to work together in teams so why teach children to work alone in the first place.

The collaboration framework on this project could also have uses in fields unrelated to live coding such as robotics and show controls. Any system that uses OSC messages as a form of communication from multiple sources could build of the implementation of the server from this project to help aid them in their development.

Chapter 6

Conclusion

6.1 Conclusion

The aim of this project was to Create a Collaborative Environment for Live Music Composition using Sonic Pi. As outlined in this report, this objective was successfully completed. A fully functional collaborative environment to allow live music composition was created allowing for multiple users to make music together via mobile device interfaces relaying the information to Sonic Pi for translation and playback. I am delighted with the successful outcome of this project and hope to build on my work to date by developing some of the goals discussed in §5.3 Future Work in particular, expanding this project to work on a global network and tackling some of the issues with inconsistency.

This project provided me with an in-depth look at the field of live coding as well as collaboration within and outside of live coding. During my research it became clear that collaboration within live coding is still a new and emerging field with many different methods of collaboration battling it out to become the de facto standard solution to the problem. My hope is that this project or at the very least the framework and design of this project can be used to help see the goal of creating a collaborative environment for live coding in a different perspective, where touch interfaces can be used as part of the ensemble. Along with this project being used to support Sonic Pi's ongoing effort to get children engaged and participating in music and coding by having them work together and showing the different avenues of computing, from UI designs, to coding in Sonic Pi, to understanding the role of a backend through the use of a server.

"A creative solution in a live coding system or a live coding performance is likely to be of an historic interest, due to how young the field is, and how extremely broad and heterogeneous the practices within it are, touching equally upon the arts and the sciences." [40]

Bibliography

[1] Sonic-pi.net. (2018). Sonic Pi - The Live Coding Music Synth for Everyone. [online] Available at: https://sonic-pi.net/ [Accessed 1 Apr. 2018].

[2] Collins, N., McLean, A., Rohrhuber, J. & Ward, A. (2003), Live Coding Techniques for Laptop Performance, Organised Sound 8(3): pp 321-30.

[3] Renate Wieser, Julian Rohrhuber (2013) Personal Accounts on the History of Live Coding Düsseldorf, DE, University of Paderborn, DE.

[4] Magnusson, Thor (2013) *The threnoscope: a musical work for live coding performance.* In: First International Workshop on Live Programming in conjunction with ICSE 2013, May 18th - 26th, 2013, San Francisco, CA

[5] Alan Blackwell, Alex McLean, James Noble, Julian Rohrhuber (2014). Collaboration and learning through live coding - Report from Dagstuhl seminar Dagstuhl, Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik, pp. 146-147

[6] TOPLAP. (2018). About. [online] Available at: https://toplap.org/about/ [Accessed 05 Apr. 2018].

[7] C.Engelbart, William K. English, *AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference*, San Francisco, CA, December Vol. 33, pp 395-410

[8] Ryan Kirkbride (2017). *Troop: A collaborative tool for live coding*, Proceedings of the 14th Sound and Music Computing Conference, July 5-8, Espoo, Finland.

[9] Digego.github.io. (2018). The Extempore programming environment Extempore 0.7.0 documentation. [online] Available at: http://digego.github.io/extempore/index.html [Accessed 6 Apr. 2018].

[10] Sorensen, A. and Brown, A. R. (2007). aa-cell in practice: an approach to musical live coding. Proceedings of the International Computer Music Conference, Copenhagen. ICMA, pp. 292-299.

[11] Andrew Schmeder, Adrian Freed, David Wessel. (2010). Best Practices for Open Sound Control Andrew Schmeder and Adrian Freed and David Wessel, Berkeley, CA, UC Berkeley.

[12] Opensoundcontrol.org. (2018). Introduction to OSC | opensoundcontrol.org. [online] Available at: http://opensoundcontrol.org/introduction-osc [Accessed 11 Apr. 2018].

[13] S. Wilson, N. Lorway, R. Coull, K. Vasilakos, and T. Moyers, (2014)"Free as in beer: Some explorations into structured improvisation using networked live-coding systems," Computer Music Journal, vol. 38, no. 1, pp. 54–64, 2014.

[14] TOPLAP. (2018). Troop: a collaborative editor for live coding. [online] Available at: https://toplap.org/troop-a-collaborative-editor-for-live-coding/ [Accessed 6 Apr. 2018].

[15] Ryan Kirkbride (2016) “FoxDot: Live coding with python and supercollider,” in Proceedings of the International Conference of Live Interfaces, pp. 194–198

[16] David Ogborn,(2016) “Live coding together: three potentials of collective live coding” Journal of Music, Technology & Education, vil. 9, no: 1,pp,17-31

[17] Ccrma.stanford.edu. (2018). JackTrip: JackTrip Documentation. [online] Available at: https://ccrma.stanford.edu/groups/soundwire/software/jacktrip/ [Accessed 10 Apr. 2018].

[18] D0kt0r0.net. (2018). Example: very long cat proposal for ICLC 2015. [online] Available at: http://www.d0kt0r0.net/teaching/proposal-iclc2015.html [Accessed 10 Apr. 2018].

[19] J.-P. and Chafe, C. (2009). JackTrip: Under the Hood of an Engine for Network Audio. Proceedings of International Computer Music Conference, Montreal, Canada.

[20] Lazzaro, J.; Wawrzynek, J. (2001). "A case for network musical performance". NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video. ACM Press New York, NY, USA. pp. 157–166.

[21] Sawchuk, A.; Chew, E.; Zimmermann, R.; Papadopoulos,C.; Kyriakakis,C. (2003). "From remote media immersion to Distributed Immersive Performance". ETP '03: Proceedings of the 2003 ACM SIGMM workshop on Experiential telepresence. ACM Press New York, NY, USA. pp. 110–120.

[22] Gu, X.; Dick, M.; Noyer, U.; Wolf, L. (2004). "NMP - a new networked music performance system". Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004. IEEE. pp. 176–185.

[23] Kurtisi, Z; Gu, X.; Wolf, L. (2006). "Enabling network-centric music performance in wide-area networks". Communications of the ACM. 49 (11): 52–54

[24] Trueman, D. (2007). Why a laptop orchestra?. Organised Sound, Princeton University, 12(02), p.171.

[25] Daniel Trueman, Perry Cook, Scott Smallwood, and Ge Wang (2006), PLOrk: The Princeton Laptop Orchestra, Year 1, New Jersey, Princeton University.

[26] Plork.princeton.edu. (2018). PLOrk: Biography. [online] Available at: http://plork.princeton.edu/bio.html [Accessed 12 Apr. 2018].

[27] GitHub. (2018). d0kt0r0/extramuros. [online] Available at: https://github.com/d0kt0r0/extramuros [Accessed 12 Apr. 2018].

[28] J. McCartney, (1996)SuperCollider: A new real time synthesis language, in Proc. International Computer Music Conference (ICMC'96), pp. 257–258.

[29] GitHub. (2018). samaaron/sonic-pi. [online] Available at: https://github.com/samaaron/sonic-pi/wiki/Sonic-Pi-Internals [Accessed 13 Apr. 2018].

[30] Tidalcycles.org. (2018). Tidal - Home. [online] Available at: https://tidalcycles.org/ [Accessed 11 Apr. 2018].

[31] Sam Aaron (2015), Programming as Performance: Live Coding with Sonic Pi, Amsterdam, OSCON 2015

[32] in_thread. (2018). Sonic Pi Architecture. [online] Available at: https://in-thread.sonic-pi.net/t/sonic-pi-architecture/329 [Accessed 20 Apr. 2018]

[33] Groups.google.com. (2018). Google Groups. [online] Available at: https://groups.google.com/forum/#!topic/sonic-pi/hTUAV1bfqls [Accessed 20 Apr. 2018].

[34] GitHub. (2018). samaaron/sonic-pi. [online] Available at: https://github.com/samaaron/sonic-pi/blob/master/etc/doc/tutorial/05.7-Thread-Synchronisation.md [Accessed 25 Apr. 2018].

[35] in_thread. (2018). Using :sync with live_loops. [online] Available at: https://in-thread.sonic-pi.net/t/using-sync-with-live-loops/172 [Accessed 25 Apr. 2018].

[36] YouTube. (2018). Sebastien Rannou, Daft Punk - Aerodynamic with Sonic Pi. [online] Available at: https://www.youtube.com/watch?v=cydH_JAgSfg [Accessed 25 Apr. 2018].

[37] YouTube. (2018). Sam Aaron live coding an ambient electro set w/ Sonic Pi. [online] Available at: https://www.youtube.com/watch?v=G1m0aX9Lpts [Accessed 25 Apr. 2018].

[38] YouTube. (2018). Sam Aaron live coding a DJ set with Sonic Pi. [online] Available at: https://www.youtube.com/watch?v=KJPdbp1An2s [Accessed 25 Apr. 2018].

[39] in_thread. (2018). in_thread. [online] Available at: https://in-thread.sonic-pi.net/ [Accessed 22 Apr. 2018].

[40] Magnusson, Thor (2013) *The threnoscope: a musical work for live coding performance.* In: First International Workshop on Live Programming in conjunction with ICSE 2013, May 18th - 26th, 2013, San Francisco, CA pp.4