


Collaborative Interfaces for Ensemble Live Coding Performance

A decorative line starts with a red dot on the left, goes down, then right, then up, then right, then down, ending with a blue dot on the right.

Ryan Philip Kirkbride

A decorative line starts with a yellow dot on the right, goes down, then left, then down, then left, ending with a green dot on the left.

Submitted in accordance with the
requirements for the degree of
Doctor of Philosophy

The University of Leeds
School of Music
December 2020

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

The right of Ryan Philip Kirkbride to be identified as Author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Acknowledgements

First and foremost I would like to thank my supervisors, Dr. Luke Windsor and Dr. Guy Brown. Their counsel and guidance has been invaluable over the course of this PhD. It was not a straightforward journey but you always supported my decisions and continuously helped develop my ideas. Thank you. Special thanks to the incredibly talented and hard-working Lucy Cheesman, Laurie Johnson, and Innocent Granger for their time and dedication as members of the The Yorkshire Programming Ensemble. This would not have been possible without your contributions. This work was conducted as part of the White Rose College of Arts & Humanities ‘Expressive Non-verbal Communication in Ensemble Performance’ network and I am incredibly grateful for the support from all of those involved. As well as my supervisors, this involved Dr. Renee Timmers, Dr. Helena Daffern, Dr. Catherine Laws, and Dr. Freya Bailes. I would also like to personally thank fellow network students, Sara D’Amario and Nicola Pennill, for the countless hours spent working together on the various network activities over the years, it was a joy to work with you. I would also like to thank Dr. Alex McLean for introducing me to live coding and inspiring me to work on my own projects, which he has continually supported. He has also given up his time and energy organising many of the live events and workshops that I took part in as part of this PhD and, if it wasn’t for people like Alex, people like me wouldn’t get the opportunity to share their own crazy ideas with the world. I am also very grateful to the live coding community for embracing my work; it has been a real driving force of motivation. I would be remiss if I did not thank my parents who, although never quite understood what it was that I was doing, were always supportive and encouraging me to pursue my passions. Thank you for everything you have done for me over the years. To my partner, Elspeth, whose strength has kept me going through all the ups and downs of this journey, you have my eternal gratitude. I could not have done this without you.

This work was supported by the University of Leeds through the White Rose College of the Arts & Humanities.

Abstract

This research is a practice-led investigation into collaborative user interfaces within the practice of live coding; the act of writing computer code for generating improvised music live in front of an audience. It examines the impact of user interface design parameters on group creativity and explores the roles of data, text, and programming languages as media for musical communication. Utilising a multi-faceted research method that combines iterative “participatory design” (Spinuzzi, 2005) with performance-led “research in the wild” (Benford et al., 2013), this research couples ethnographic and autoethnographic observations to gain insight into the practice of ensemble live coding and inform software design. Three novel collaborative interfaces have been developed as part of this research that explore various facets of musical collaboration in live coding. Each interface was developed through an iterative and reflexive methodology focused on user-centred design and was employed in a cyclical process of artistic practice and refinement based on user evaluation and in-depth study. The first interface, entitled Troop, is a shared text editor that allows multiple performers to collaborate on the same single body of code together. The second, CodeBank, explores how private working in a collaborative context affects creativity and improvisation. Finally, PolyGlot, combines multiple live coding languages into a single collaborative interface that enables live coding musicians to play together, regardless of their knowledge of languages. As well as these three graphical interfaces, the functionality of an existing live coding language, FoxDot, was extended to help facilitate the sharing of musical information within an ensemble. Each interface was used in live performance by The Yorkshire Programming Ensemble and evaluated through group interview sessions that examined the themes of immediacy, trust, and risk with regards to both human-computer interaction and intra-ensemble communication as well as the experience of personal- and group-flow states.

Contents

Acknowledgements	ii
Abstract	iii
Table of Contents	iv
Appendices	viii
Recording Documentation	ix
List of Figures	x
1 Introduction	1
1.1 Context	1
1.2 Method	2
1.3 Outcomes	3
1.4 Thesis outline	4
2 Contextual Background	6
2.1 What is Live Coding?	6
2.1.1 Definition	6
2.1.2 Programming as performance	7
2.1.3 The TOPLAP manifesto draft	8
2.1.4 Show us your screens	9
2.1.5 Improvisation	10
2.1.6 Existing technologies for live coding	11
2.2 Collaboration and Network Music in Live Coding	15
2.2.1 Network music systems	15
2.2.2 The role of the network in network music	18
2.2.3 Network music systems for collaborative live coding	20
2.2.4 Futures of live coding collaboration	25
3 Method	27
3.1 Introduction	27
3.2 Rationale for Research	27
3.3 Methodology	29
3.3.1 Research in the wild	29
3.3.2 Participants	33

3.3.3	Considerations	34
3.4	Chapter structure	35
4	Foundation Work	36
4.1	Introduction	36
4.2	FoxDot	36
4.2.1	Player objects	37
4.2.2	Patterns	39
4.2.3	Time-dependant variables	41
4.3	Considerations	42
5	Troop: An Interface for Real-Time Collaborative Live Coding	43
5.1	Introduction	43
5.2	Motivation	43
5.3	Phase 1: Inital Implementation	46
5.3.1	Development	46
5.3.2	Practice	50
5.3.3	Evaluation and outcomes	51
5.4	Phase 2: Operational Transformation	53
5.4.1	Development	53
5.4.2	Practice	55
5.4.3	Evaluation and outcomes	57
5.5	Phase 3: Language Agnosticism	59
5.5.1	Development	59
5.5.2	Practice	61
5.5.3	Evaluation and outcomes	63
5.6	Conclusions	64
5.6.1	Personal reflection	64
5.6.2	User evaluation	66
5.6.3	Quantitative evaluation	68
5.6.4	Impact	70
5.6.5	Potential in pedagogy	71
5.6.6	Final thoughts	72
6	Developing a Language for Live Coding in Ensemble Performance	73
6.1	Introduction	73
6.2	Phase 1: Modelling Interpersonal Musical Relationships	74

6.2.1	Development	74
6.2.2	Practice	75
6.2.3	Evaluation and outcomes	76
6.3	Phase 2: Player-Key Data Structures	77
6.3.1	Development	77
6.3.2	Practice	83
6.3.3	Evaluation and outcomes	84
6.4	Phase 3: Extending Player-Keys for Musical Behaviours	87
6.4.1	Development	87
6.4.2	Practice	91
6.4.3	Evaluation and outcomes	92
6.5	Conclusions	94
6.5.1	Personal reflection	94
6.5.2	User evaluation	97
6.5.3	Final thoughts	102
7	CodeBank: Public and Private Working in Ensemble Live Coding	104
7.1	Introduction	104
7.2	Motivation	104
7.3	Phase 1: Initial Implementation	106
7.3.1	Development	106
7.3.2	Practice	110
7.3.3	Evaluation and outcomes	111
7.4	Phase 2: User Experience	112
7.4.1	Development	112
7.4.2	Practice	114
7.4.3	Evaluation and outcomes	115
7.5	Phase 3: Synchronisation and User Monitoring	117
7.5.1	Development	117
7.5.2	Practice	120
7.5.3	Evaluation and outcomes	120
7.6	Conclusions	122
7.6.1	Personal reflections	122
7.6.2	User evaluation	125
7.6.3	Final thoughts	130

8	Polyglot: A Multilingual Interface for Collaborative Live Coding	132
8.1	Introduction	132
8.2	Motivation	132
8.3	Phase 1: Initial Implementation	134
8.3.1	Development	134
8.3.2	Practice	136
8.3.3	Evaluation and outcomes	137
8.4	Phase 2: Language-Specific Feedback	138
8.4.1	Development	138
8.4.2	Practice	139
8.4.3	Evaluation and outcomes	140
8.5	Conclusions	143
8.5.1	Personal reflection	143
8.5.2	User evaluation	145
8.5.3	Final thoughts	148
9	General discussion and conclusions	150
9.1	Introduction	150
9.2	Timeline	150
9.3	Discussion	151
9.4	Addressing the Research Questions	153
9.5	Conclusion	162
	Appendix A: Performance Descriptions	175
A.1	Leeds Algorave, Open Data Institute, Leeds - 28/04/17	175
A.2	Algorave Assembly Lunchtime Concert, Leeds - 27/04/18	177
A.3	International Conference on Live Interfaces, Porto - 14/06/18	181
A.4	Rehearsal session, various locations - 26/04/17	183
A.5	Rehearsal session, various locations - 06/06/17	185
A.6	Together In Music conference, York - 14/04/18	186
A.7	Algo-Rhythms, Rotterdam, 28/04/2019	188
A.8	Rehearsal session, Sheffield - 09/12/18	191
A.9	TOPLAP End of Cycle Party, Access Space, Sheffield - 19/12/18	194
A.10	Late at the Library: Algorave, London - 05/04/19	197
A.11	Rehearsal session, Sheffield - 07/05/19	202
A.12	AlgoMech Festival, DINA Club, Sheffield - 18/05/19	204

Appendices

Included in this thesis:

Appendix A. Written performance descriptions.

Included as files accompanying this thesis:

Appendix B. Troop project files.

Appendix C. Troop user evaluation questionnaire responses.

Appendix D. FoxDot project files.

Appendix E. CodeBank project files.

Appendix F. Polyglot project files.

Recording Documentation

Included as files accompanying this thesis.

Recording A. ch5_1a-Rehearsal-09_04_17.mpg

Recording B. ch5_1b-Leeds_Algorave-28_04_17.mp4

Recording C. ch5_2-Algorave_Assembly-27_04_18.avi

Recording D. ch5_3-ICLI-14_06_18.mp4

Recording E. ch6_1a-Rehearsal-26_04_17.mpg

Recording F. ch6_1b-Rehearsal-06_06_17.mpg

Recording G. ch6_2-Together_in_Music-14_04_18.mpg

Recording H. ch6_3-Algo_Rhythms-28_04_19.mpg

Recording I. ch7_1-CodeBank_Rehearsal-09_12_18.mpg

Recording J. ch7_2-TOPLAP_End_of_Cycle_Party-19_12_18.mpg

Recording K. ch7_3-British_Library_Algorave-05_04_19.mpg

Recording L. ch8_1-Rehearsal-07_05_19.mov

Recording M. ch8_2-AlgoMech_Festival-18_05_19.avi

List of Figures

2.1	Comparison of SuperCollider and ixi-lang syntax	12
2.2	TidalCycles code for a polyrhythmic drum beat	13
2.3	Example of a Sonic-Pi live loop	14
2.4	Network Music classification graph	15
3.1	Overview of performance-led research in the wild	32
4.1	Example of typical FoxDot code.	37
4.2	Comparison of FoxDot code with Python’s standard library.	38
4.3	FoxDot code for playing multiple notes simultaneously.	38
4.4	Example FoxDot code using the <code>play</code> SynthDef.	39
4.5	Example FoxDot code using the <code>every</code> method.	39
4.6	FoxDot <code>Pattern</code> transformations and output.	40
4.7	Comparison of transforming FoxDot <code>Pattern</code> objects and Python lists.	40
4.8	Example <code>Pattern</code> functions used to generate sequences.	41
4.9	Example FoxDot code using a <code>TimeVar</code>	42
4.10	Example FoxDot code using a <code>TimeVar</code> shared between two player objects.	42
5.1	Early version of the Troop interface.	47
5.2	Final design of the Troop interface.	48
5.3	Troop’s Network diagram.	49
5.4	Example of scrambled text in Troop.	49
5.5	Screenshot of inconsistent text contents across Troop clients.	50
5.6	The Yorkshire Programming Ensemble. Photo by Aaron Ratcliffe.	51
5.7	Representation of multicoloured text in Troop using user ID numbers.	54
5.8	Python Code for calculating a user’s location after a text operation.	55
5.9	Troop interface with transparent background.	56
5.10	Still frame from the Algorave Assembly performance	56
5.11	Troop interface overlaying on SuperCollider’s oscilloscope.	58
5.12	Windows used to start a colour merge sequence in Troop.	60
5.13	Progression of Troop’s font colour merge.	61
5.14	Class Compliant User Interfaces using Troop	64
5.15	User satisfaction response graph	69

5.16	Photos of Troop being used by various ensembles.	70
5.17	Troop used at Tidal Club, Sheffield.	71
6.1	FoxDot code using the <code>follow</code> method	75
6.2	Screenshot from a rehearsal session.	76
6.3	FoxDot code using the <code>follow</code> method with chords.	76
6.4	FoxDot code using the player-key data structure.	78
6.5	Flow chart diagram of psuedo-reactive player-key data type.	78
6.6	Tree structure relationships of parent and child player-key data structures	79
6.7	Python code for player-key operator overloading	80
6.8	FoxDot code using player-keys to select single notes from a chord.	80
6.9	FoxDot code using player-keys and a logical “equals to” test.	81
6.10	FoxDot code using player-keys and a logical “greater than or equals to” test.	81
6.11	FoxDot code combining multiple logical tests with the same player-key.	81
6.12	Code for the player-key <code>map</code> method.	82
6.13	FoxDot code using the player-key <code>map</code> method.	83
6.14	FoxDot code using functions as input for the player-key <code>map</code> method.	83
6.15	FoxDot code using player-key to map pitch to panning.	84
6.16	Circular reference error.	84
6.17	FoxDot code using the player-key <code>transform</code> method.	88
6.18	FoxDot code using the the player-key <code>accompany</code> method.	89
6.19	Code for the <code>versus</code> method from the player object class.	90
6.20	FoxDot code using the <code>versus</code> method.	91
6.21	Photo from Algo-Rhythms performance	92
7.1	CodeBank’s Network diagram.	106
7.2	Screenshot of the CodeBank client interface.	107
7.3	Close up of the CodeBank Code Repository	108
7.4	Action Buttons used for interacting with the CodeBank interface	109
7.5	Screenshot of the CodeBank server interface	110
7.6	Comparison of headphone use over the course of a CodeBank session.	112
7.7	Screenshot of the updated CodeBank client application and chat box.	113
7.8	Window for adjusting the beat in FoxDot from CodeBank.	114
7.9	Photo from the End of Cycle performance.	115
7.10	Frame from video of Sheffield Algorave performance with CodeBank.	117
7.11	User monitoring functionality of CodeBank	118
7.12	CodeBank chatbox with flashing borders.	119

7.13	Login window for CodeBank with TidalCycles language option.	119
7.14	Photo from Late at the Library.	121
8.1	Polyglot network diagram.	134
8.2	Login window for the Polyglot interface.	135
8.3	Photo of Polyglot being used with 4 users.	136
8.4	Screenshot from a Polyglot rehearsal recording.	137
8.5	Updated login interface for Polyglot	139
8.6	Screenshot of the AlgoMech performance with the blurring text cursors.	142
8.7	Polyglot interface showing feedback on an error in the SuperCollider tab.	144
9.1	Project gantt chart	151
9.2	Spectrum of affordance for a live coding interface framework and a collaborative live coding interface framework.	153
9.3	Examples of comments being used during performance.	160

1. Introduction

1.1 Context

Live coding is a relatively young and interdisciplinary, primarily musical, performance practice that uses computer programming as its medium. Performers generate music through the construction and reconstruction of algorithms using programming languages while projecting their screens for the audience to see (Mori, 2015). It emerged at the turn of the millennium and is still a relatively unexplored area of research (Burland and McLean (2016) and Magnusson (2014) for example). It crosses the disciplines of music and computer science, as well as their many sub-disciplines, and explores the act of composition as performance. The practice of live coding is not restricted to just the creation of music but can also be used to generate visuals, using such environments as fluxus¹ and livecodelab², and even create interactive dance choreography (Sicchio, 2014).

Live coding is very often enacted as a solo performance, but why is this? It might be due to the complicated nature of setting up collaboration software and synchronising computer clocks but may also be because of the nature of the style of performance itself. Working together with other people to compose a constantly evolving piece of music in real-time may not only give rise to artistic and social conflicts but would be a cognitively challenging task to begin with. The issue is further confounded by the fact that the network of live coders is spread out so wide that many practitioners are simply too estranged from one another to develop fruitful artistic relationships. Research suggests that “being a member of a music ensemble can enhance subjective wellbeing, support the development of musical identity and a sense of purpose” and traditional types of group music making, such as singing, “can foster happiness as well as provide musical and social benefits” (What Works Wellbeing, 2016). Computer programmers are often associated with the stigma as the lonely introvert working in solitude in a badly lit room and this image would not be inaccurate when describing a live coding event. Unfortunately, there is no “industry standard” for ensemble live coding and this leaves many practitioners without access to an aspect of musical performance that can benefit their wellbeing and mental health. This is not to say that there are no existing musical collaborations in the community of live coding but many live coding ensembles consist of performers based in research at the cutting edge of the practice such as OFFAL³, BEER⁴, and the Cybernetic Orchestra⁵.

¹<http://www.pawfal.org/fluxus/>, accessed: 28/05/19

²<https://livecodelab.net/>, accessed: 28/05/19

³<https://offal.github.io/>, accessed 18/10/20

⁴<http://www.beast.bham.ac.uk/offspring/beer/>, accessed 18/10/20

⁵<https://global.mcmaster.ca/activity/cybernetic-orchestra/>, accessed 18/10/20

A common practice in live coding is to project your screen so that the audience can see exactly what you are doing. Too often the live coder is seen to be static behind the glow of their laptop with a look of intense concentration on their face, so sharing the contents of the screen is the main medium of communication between performer and audience. Performers' movements are usually limited to just fingers typing on keyboards and viewers are unable to connect any meaning between movement and sound so the projected code is the only means of linking what is seen and what is heard. But programming languages are just that; languages. They don't make sense without an understanding of its lexicon and syntax, and some audience members are left severely disadvantaged when this is the case. With these language barriers in place it begs the question, are audiences able to interpret collaborative musicianship in ensemble live coding performances?

The prevalence of internet-based technologies has enabled live coders to take part in networked performances that allow performers to play together from geographically separate locations. Consequently, a live coding performance does not always require a performer to be physically present with the audience in a venue. For ensembles, this also means communication between performers can be limited as they are not able to see each other during a performance. Many groups will make use of an instant-messaging application, which raises the question; how does a separate channel of communication affect ensemble performance? Not all live coding ensembles perform from different locations, however, and many perform together on stage. Here, communication is no longer limited to a digital channel but can also be realised physically; performers can look at each other, nod and wave, and even dance with one another. Does the physical presence of performances affect audience perception or is it just the music that counts? The questions raised here are just a few that this body of work will explore with the objective of developing more engaging methods of practice for live coders. While the primary goal of this work is to better facilitate communication and collaboration in live coding, it is also an exploration of methods for engaging the public with a new and exciting performance practice.

1.2 Method

Practice-based, or artistic, research is research that takes place “in and through art practice” such that the “artistic practice is not only the result of the research, but also its methodological vehicle” (Borgdorff, 2010, p. 46). With regards to this thesis, the focus will not just be on the artistic *outcomes* of ensemble live coding, i.e. the performances, but also the encompassing process of developing the tools and craft en route to reaching these outcomes. This research will use a mixed-methodology approach, combining practice-based research with ethnographic studies to create new performance software as well as develop tacit knowledge and produce novel musical performances that explore multiple facets of collaborative live coding.

The development of the software will also be largely informed through the study of live performances, a method that shares many similarities with the performance-led “research in the wild” methodology described by Benford et al. (2013). They suggest that performance-led research is based on the foundation of three activities; practice, study, and theory. While Benford et al. (2013) primarily consider the act of performance to be the practice, this thesis is embedded in practice-based research, which, as mentioned above, considers practice to consist of both the artistic outcomes and their methodological vehicles. Here the practice involves the development of software for collaborative live coding as well as its application in live performance contexts. These performances will be documented and studied using an ethnographic approach, which will help inform future design decisions. The development of these interfaces themselves will be done iteratively over the course of several “design phases” and follow guidelines for software design outlined by Norman (1998). In each design phase, the results from studying the performance and user feedback will be used to improve the software going forward. This methodology, known as “participatory design” (Spinuzzi, 2005), frames design as the research itself in that the design leads to the production of artefacts, systems, and practical or tacit knowledge. After the final design phase, more in-depth reflection on the success and failures of the interface will take place, using guidelines for evaluating software for musical improvisation outlined by Gifford, Knotts, Kalonaris, and McCormack (2017).

1.3 Outcomes

The key contributions to knowledge from this thesis are:

1. The collaborative live coding environment, *Troop*, that allows multiple performers to work on the same code together simultaneously (Chapter 5).
2. The development of the live coding language, *FoxDot*, to become a vehicle for ensemble communication and the creation of dynamic musical relationships (Chapter 6).
3. The *CodeBank* performance system, which splits live coding performance into public and private activities to encourage experimentation and improve musical output (Chapter 7).
4. The multilingual collaborative live coding environment, *Polyglot*, that allows users to collaborate using up to three different live coding languages simultaneously (Chapter 8).
5. A series of recorded and transcribed ensemble performances using the software above (Appendix A).

1.4 Thesis outline

While the current chapter briefly describes the context of the problem being addressed by this research, Chapter 2 does so in more detail. It provides a contextualisation of this PhD with regards to the practice live of live coding and its ties to improvisation as well as the history of musical collaboration over a computer network as a whole. It also discusses a wide range of technologies used by existing live coding ensembles, both technologically homo- and heterogeneous. With this research properly contextualised, Chapter 3 goes on to discuss its methodology, which includes the research questions and the rationale behind them. It outlines the theories used to design, develop, and critically reflect upon the use of any collaborative interfaces that will be developed to address the research questions and discusses the challenges that will likely be faced as a result of these choices.

Several pieces of software have been developed that facilitate collaborative live coding in different ways and each is discussed primarily within their own chapter. Of these software outputs there are three graphical user interfaces and a live coding language that supports inter-musician communication through code. The latter is called FoxDot, which has been in development since 2015 but was primarily designed for solo performance. Chapter 4 outlines the foundation work on FoxDot that has taken place prior to the start of this PhD research and should provide the reader with enough information about its features and syntax to understand the technical discussion in subsequent chapters. The first graphical user interface developed is Troop, which is a shared live coding text editor in which all participating users edit the same body of code together simultaneously. Each user has a labelled cursor and a different coloured font to help identify each users' contributions. This not only helps users see the ongoing actions of their co-performers but also gives audiences insight into the collaborative processes as the different coloured text become interweaved over time. As a direct result of the research into the Troop editor, several features were added to the FoxDot language that allow users to share musical information and create dynamic musical relationships within their code. Chapter 6 documents the process of adding these features and improving FoxDot as a vehicle for ensemble communication. Chapter 7 introduces the CodeBank system, which gives performers a private workspace to test ideas using code and listen to the results before the audience. Instead of adding code to a single body of text, it is represented as a repository of small chunks of code, known as "codelets", that can be opened in any user's private workspace and edited. This aims to give users the confidence to experiment more, while also providing audiences with the most polished musical experience possible. The last interface developed is Polyglot, which is discussed in Chapter 8. Polyglot uses much of the source code from the Troop interface but extends its functionality to utilise three live coding languages instead of one. The interface incorporates multiple text boxes that allow concurrent editing, each one

connected to a separate live coding language. Users given a labelled cursor and different coloured font to help performers and audience members alike track the changes to the code over time.

Each of these chapters are accompanied by video documentation of rehearsals and performances using the different software at each design phase. The videos themselves are transcribed and discussed with the aim of developing the knowledge to be able to answer the research questions posed in this PhD thesis. The final chapter, Chapter 9, addresses these questions using data gathered throughout the thesis.

2. Contextual Background

2.1 What is Live Coding?

2.1.1 Definition

Before answering the question of “how can the practice of ensemble live coding be improved?” we must first define what that practice consists of. The earliest documented live coding performance took place in 1985 by Ron Kuivila (TOPLAP, 2004) but it wasn’t until the turn of the millennium that live coding appeared as we know it today. One of its earliest descriptions can be found in (Collins, McLean, Rohrhuber, & Ward, 2003) in which it is described as “coding music on the fly” and “tweaking or writing the programs themselves as they perform”. Since then, the popularity of live coding has risen and many new practitioners and technologies have shaped the practice as it has evolved. The following is a more recent definition, taken from the TOPLAP¹ website, which was created in 2004 to help promote the practice of live coding:

Live coding is a new direction in electronic music and video, and is getting somewhere interesting. Live coders expose and rewire the innards of software while it generates improvised music and/or visuals. All code manipulation is projected for your pleasure. Live coding works across musical genres, and has been seen in concert halls, late night jazz bars, as well as algoraves. There is also a strong movement of video-based live coders, writing code to make visuals, and many environments can do both sound and video, creating synaesthetic experiences.

In layman’s terms, live coding is the creation of music and/or graphics through the use of computer programming languages live in front of an audience, where the program generating the output is edited by the performer while it is still running. This is a stark contrast to more traditional computer programming where code is saved, compiled, and then run in its entirety. It has a strong focus on the visual elements of performance, with some live coders only creating graphical output, and all code being projected for the audience’s pleasure. Often performers start from a “blank slate” and write all of the code for generating music during the performance (Brown & Sorensen, 2007; Magnusson, 2011a).

Computers are very good at running many simple commands over and over again in a loop and this behaviour lends itself well to the creation of dance music, which is why live coding is often

¹TOPLAP stands for Temporary Organisation for the Promotion of Laptop Algorithm Performance - see <http://toplap.org/>

experienced at late-night club events known as *Algoraves*² (short for Algorithmic Rave). This is not to say that live coding can only be used to perform electronic dance music, but it is very good at it. Live coding is not just limited to music either. As mentioned previously it can be used to generate real-time graphics and has also been used to live choreograph dancers (Sicchio, 2014). Live coding is still a developing practice and has been used within a wide variety of styles and genres of music, with many more still to come.

2.1.2 Programming as performance

A live coder is someone who writes computer programs that generate music; so are they performers or are they composers? Or are they something else entirely? The ambiguity here brings into question the role of the live coder within the context of traditional western music and the composer-performer relationship. Many researchers feel that “live coding is, in many ways, more similar to musical composition than to instrumental performance.” (Aaron, Blackwell, Hoadley, & Regan, 2011). This sentiment is shared by Magnusson (2011a), who suggests that “the live coder is primarily a composer, writing a score for the computer to perform” but the act of composition is, in and of itself, the performance. Similarly, Blackwell and Collins (2005) state that “live programming includes notation, but the notation is ‘performed’ automatically by the computer, without error”. There is no such thing as “wrong notes” for a computer but a live coder may enter values in their code that produce less than satisfactory results but are still embedded within the overall sonic experience of the performance; “We are observing a kind of ‘live studio’ composition where mistakes – perhaps there are no such thing, just consequences – cannot be ‘corrected’” (Emmerson, 2007, p. 113). Very often live coders will choose to “purposely embrace error and failure” (McLean, 2017) in their performance, which brings to mind one of Brian Eno’s oblique strategies; “honour your error as the hidden intention” (Garland, 2001). It suggests that the indeterminacy of one’s own actions should be incorporated into practice as opposed to being avoided. Computer programs (especially those that are written in front of an audience) are also susceptible to bugs and errors themselves and these can occur during a live performance. For live coders it is often about “allowing those errors and glitches to occur in a context that works aesthetically” (Collins et al., 2003); a philosophy sometimes known as “errorancy” (Cascone, 2011) where errors act as divine intervention in artistic practice.

Due to the risks involved when working glitch and error, live coding is a cognitively demanding task and is as much about “braving the challenges of coding music on the fly” (Collins et al., 2003) as it is about performing music. The high level of difficulty is part of live coding’s appeal and by deciding to live code, live coders choose “to embrace the challenge of live coding; the virtuosity of the required cognitive load, the error-proneness, the diffuseness, all of these play-up

²<http://algorave.com/>, accessed 06/11/2018

the live coder as a modern concerto artist” (Blackwell & Collins, 2005). Things can, of course, go wrong during performance but “recovery’ is a skill in itself as all the best performers take risks” (Emmerson, 2007, p. 112) and by embedding risk within their practice itself, live coders attempt to elevate themselves to a higher echelon of musical performance. The significance of error and failure in live coding performance places the role of the live coder in a unique juxtaposition between composer and performer to create an artistic practice that is a “fascinating fusion of composition and improvisation” (Freeman & Van Troyer, 2011).

2.1.3 The TOPLAP manifesto draft

The TOPLAP Manifesto Draft (TOPLAP, n.d.) was developed with the aim of providing a set of rules for live coding performances and is often referred to when attempting to define the practice; as done so by Burland and McLean (2016); Flašar (2016); Magnusson (2014); Sicchio (2014); Swift, Sorensen, Martin, and Gardner (2014) among others. It makes several demands, most notably “Obscurantism is dangerous. Show us your screens”, which have heavily impacted the practice of live coding. Many defining qualities of live coding, such as screen projection, improvisation, and embracing error, are mentioned in the manifesto and are likely responsible for shaping the practice into its current state. The concept of live coding as a challenge is at the heart of the manifesto and this is exemplified well in two lines: “[We prefer] The skillful extemporisation of algorithm as an expressive/impressive display of mental dexterity” and “No backup (minidisc, DVD, safety net computer)”. This puts a focus on improvisation without any backup or safety reset, which would likely result in mistakes and even program crashes. Perhaps this is the origin of the live coding culture choosing to “embrace error and failure”. In an era of pre-recorded performances and auto-tune, there is clearly a vision in this manifesto for pursuing a sense of “liveness” that pushes against even the ideas of improvisation being restricted by musical conventions (Auslander, 2008, p. 64). Like the Art of Noises manifesto (Russolo, 1913), the TOPLAP Manifesto Draft aims to guide artists into a new manner of performance practice. Where Russolo had the bold new idea that noise could be music, TOPLAP promotes the idea of the algorithm as an instrument intrinsically tied to a human performer and pushes against the ubiquity of artificial intelligence in the 21st century. Of course, there are problems with manifestos and the reality of the implementation does not always align with the theory that it was built upon. Perhaps the most famous example of this is the communist manifesto and its failed realisation within the Soviet Union. Forcing the idealism of a manifesto upon people will always result in push back and in the field of live coding there are already performers choosing to hide their screens in protest and pre-write their code just because they can. A manifesto is a vision, but not everyone’s vision. This is partly why the TOPLAP manifesto is still labelled as a draft; it is a definition open to change.

2.1.4 Show us your screens

The projection of the performer’s screen is a practice that is commonplace in live coding (Mori, 2015) and is *demande*d of the performer as part of the TOPLAP Manifesto Draft. This is just one of the ways that live coding differs to other types of musical performance because “in most other musics the score is hidden from the audience (it is either visible to the performer/s only, or is memorised in advance of the performance) and therefore the process of musical creation is partially hidden” (Burland & McLean, 2016, p. 2). Unlike other performance practices live coding exposes the processes at play through the act of projecting one’s screen and gives the audiences an insight into what is going on. In fact, there are multiple roles for the screen that can be considered:

Primarily it is used by the performer to construct the performance, but importantly it is also often a key part of the performance. Observing how the code changes and its relationship to the music can elicit an aesthetic response just as can the auditory component of the music. Moreover it can help the audience appreciate simply that it is possible to create such a performance live, which they may not have previously considered. Finally, the code may also be a (partial) record of the performance, useful for teaching or other communicative activities after the performance. (Blackwell, McLean, Noble, & Rohrhuber, 2014, p. 139)

Furthermore, the screen also provides a level authenticity to the live performance: it acts as proof that the performer is not just “pressing play” and then checking their emails for the remainder of a performance (Zmölnig, 2016). A common criticism of electronic dance music (EDM) and, more broadly, modern pop music is the apparent lack of authenticity; performers seemingly rely on auto-tune and drum-loop software to create songs, and performances tend to be closer to pre-recordings than live music. This opinion is well exemplified in an online article from 2012 that surfaced after reality TV star, Paris Hilton, made her debut as a DJ (Mann, 2012). While the views of Mann are not universally shared, the disconnection between an artist’s physical actions and musical output in EDM performances have proliferated the misconception that electronic music is simply pressing play. The projection of the screen in live coding alleviates this problem by giving audiences a deeper insight into the processes by showing every single mouse-click and key-press that occurs during a live performance. The use of the computer keyboard requires both hands to be used almost constantly and a high level of attention given to the computer screen. As with most laptop performances, this results in a lack of bodily movement and can be often be perceived as detrimental to the overall performance. However, projecting a performers’ code for the audience can compensate for this. A recent review of live coder Joanne Armitage’s performance described the use of screen projection as “highly effective, not least of all in that it’s an innovative way of providing a visual element to the standard person standing at a laptop performance” (Nosnibor,

2016).

Magnusson (2011a) states that the code itself is also “a representation of what occurs in the sonic domain” and can help audience decipher the complex musical patterns they are listening to. This may not always be the goal for performer, but for an audience member who is eager to learn about the practice of live coding it is quite essential. For some the screen projection is the primary source of engagement. One member of a survey of live coding audiences said that “[the code] must be shown. If not I find these events to get boring quickly because the generated music usually has little change over time” (Burland & McLean, 2016). However, another survey respondent felt that the large focus on the code had a negative impact on their experience: “Mostly I am annoyed by the visual display as it pulls the focus away from the human performers and the listening”. There is clearly a disparity in opinion and perception of projected code in live coding performance. “If the code takes on the role of ‘performer’ in live coding events then the way in which that is accessible and visible becomes crucial in order for the audience to have an optimal experience.” (Burland & McLean, 2016). One way of improving audience experiences of live coding is attempting to engage with the audience either by using text to communicate directly, or other visualisation techniques (McLean, Griffiths, Collins, & Wiggins, 2010) but more often than not a live coder will opt for a raw display of their code.

The projected code is arguably a “representation of the performer” (Burland & McLean, 2016) that is undergoing an embodiment of the programmer’s thoughts. Live coding is a performance of mental dexterity after all; music is not created by movements of the body, but the organisation of one’s thoughts. In this sense the screen becomes an extension of the body (Mori, 2015); or, even, the mind. Thinking of the performer in this context also changes the relationship with the code from an audience’s perspective:

In this context the audience themselves, rather than the programmer, might be regarded as the “end-user”. The audience are not producing the code, but they are consuming it. But without knowledge of the language, their consumption even of executable code can be considered as secondary notation. This is an unusual perspective from which to view code documentation, but one that may become increasingly common in fields where descriptions of software artefacts are shared between non-programmers. (Blackwell & Collins, 2005, p. 9)

2.1.5 Improvisation

When positing the question, “why would anyone want to live code music?” Collins et al. (2003) state that “live coding allows us to keep a sense of challenge and improvisation about electronic music-making”. Live coding becomes rather pointless if not for its improvisational aspect, as it

would no longer be “live” and the music could be more effectively composed using other, non-text-based applications. Improvisation in music allows performances to be catered to the mood of the room in which it takes place. In live coding “the music is often improvised, created in the moment, and the performers’ awareness of their surroundings can have an impact on the way in which the performance unfolds” (Burland & McLean, 2016). Live coding is, in many ways, composition as live performance, but shares many similarities with jazz music, which is famed for improvisation, as opposed to traditional music composition (Aaron et al., 2011). As Steve Lacy says, in free jazz, “the music always has to be – on the edge – in between the known and the unknown” (Bailey, 1992, p. 54) and the element of “risk and uncertainty adds excitement” to an improvised performance (Burland & Pitts, 2012). This also holds true for live coding; things can even go wrong to the extent that programs crash and performances are required to start over.

There are a number of ways to approach a live coding performance and one of those is to start with the proverbial ‘blank-slate’ (Brown & Sorensen, 2007; Magnusson, 2011a), wherein the live coder creates a musical program in its entirety in front of an audience. This can be a drawn out process, but it is nevertheless a demonstration of virtuosity and skill. This is not always the case, however, and different performers have different approaches to the way they make music. In the advent of newer, more terse live coding languages this process has become a much more viable performance practice and no longer sacrifices large amounts of time for typing.

2.1.6 Existing technologies for live coding

There are many languages used by a variety of live coders and each one has its own unique style, affordances, advantages, and disadvantages. It is beyond the scope of this research to discuss every live coding system that exists but this section will outline a few of the more notable languages and environments in an effort to portray the rich and diverse landscape of live coding technologies.

Possibly the most widely used environment is SuperCollider (McCartney, 2002), which is both a live coding language and audio synthesis engine that makes up the foundation for several other notable live coding languages such as TidalCycles (McLean, 2014), *ixi-lang* (Magnusson, 2011b), and the present author’s own Python-based live coding environment, *FoxDot* (Kirkbride, 2016), which is discussed in more detail in Chapter 4. SuperCollider is a powerful digital signal processing (DSP) environment that gives users access to a range of oscillators which can be patched together using code in real-time, like a text-based abstraction of a modular synthesiser. It also contains many functions and routines that allow users to schedule events in the future, such as the occurrence of notes, and many numerical patterns that can be easily transposed and transformed while being played. SuperCollider uses a client-server model to separate the language and the audio synthesis; the server, an application called ‘*scsynth*’, is dedicated to generating sound and the client, called ‘*sclang*’, sends trigger messages using Open Sound Control (OSC) (Wright, Freed,

et al., 1997). Multiple clients can send messages to a single server, which makes collaborative networked performances relatively simple, and several libraries have been written to extend the capabilities of networked collaboration in SuperCollider, such as tempo synchronisation and data sharing (discussed in Section 2.2.3).

It is logical to follow on from SuperCollider with *ixi-lang* (Magnusson, 2011b); a mini-language written in the SuperCollider language, ‘sclang’, for quick and easy live coding. This is not to say that *ixi-lang* is any less powerful or less expressive than SuperCollider; in fact it has access to all of SuperCollider’s functionality but provides a layer of abstraction that makes live performance more viable. *ixi-lang* uses a “a highly simple syntax with strong expressive constraints” designed to be “easily understandable by the audience who would be able to follow each step of the performance”. Figure 2.1 is a comparison of SuperCollider and *ixi-lang* code that are performing almost identical actions. Both sets of code perform a short repeated melody using a “SynthDef” (a pre-defined digital instrument created using combinations of oscillators and wave envelopes) called ‘piano’. SuperCollider requires explicit instructions and program constructs to perform this routine whereas *ixi-lang* does the same thing in only a few characters of code. An agent, in this case ‘agent1’, is assigned a series of notes to play using square brackets and derives its pitch and duration from the numbers and the white-space in the square brackets. Not only does this save the performer from typing separate instructions for pitch and duration, but also gives audience members a visuospatial relationship between code and sound. Agents are quantised automatically (as is the norm in most traditional western music), which also saves the performer valuable seconds in extra typing. However, this is not necessarily the desired outcome; Magnusson himself said he “became tired of the strongly timed structure of *ixi-lang* performances” and suggests that there is a trade-off between flexibility and simplicity within the language.

<pre style="background-color: #2e3436; color: #eeeeec; padding: 10px;">Pdef(\agent1, Pbind(\instrument, \piano, \degree, Pseq([2,4,7,6,7],inf), \dur, Pseq([1/2,1,1/2,1,1],inf),)).quant_(~tempo);</pre>	<pre style="background-color: #2e3436; color: #eeeeec; padding: 10px;">agent1 -> piano[2 4 7 6 7]</pre>
(a)	(b)

Figure 2.1: Comparison of (a) SuperCollider and (b) *ixi-lang* syntax

With regards to timing in live coding, perhaps one of the most interesting languages to discuss is TidalCycles, or Tidal for short. Like *ixi-lang*, it is a mini-language embedded within another programming language although, in this instance, it is based in the functional programming language, Haskell³. Tidal uses functions of time to create musical events from cyclical pattern structures. As

³<https://www.haskell.org/>, accessed 06/11/2018

musical patterns change throughout a performance, Tidal uses time to calculate what the current musical event is as opposed to re-calculating the entire performance up to that point. On top of this, TidalCycles uses a notion of rational time such that durations for musical events and patterns can be subdivided by any rational number as opposed to powers of two as in traditional western music. Figure 2.2 is an example of TidalCycles syntax that plays two rhythms simultaneously; the first is a bass drum, followed by a hi-hat, and then a snare and the second is made up of two claps. The “sound” function converts these names of audio samples into sound and also parses the text for rhythmic information. Separating groups of samples by a comma within square brackets tells the language to play each group within the same duration. The bass drum, hi-hat, and snare samples are played at “a third of a cycle” each and the claps at “half a cycle”, creating a very simple polyrhythm.

```
d1 $ sound "[bd hh sn, cp cp]"
```

Figure 2.2: TidalCycles code for a polyrhythmic drum beat

The exploitation of time is common in live coding and one of the more novel methods of this is temporal recursion. In computer science recursion is the process of telling a function to call itself and is a powerful technique in solving complex mathematical problems. A temporal recursion does the same thing, but schedules the self-referential call in the future. Recursion is not a new concept (Dijkstra, 1960) but the use of temporal recursion may only have been around since 1996 and is still not a widely used practice (Sorensen, 2013). Some examples of live coding languages that incorporate this technique are Impromptu (Sorensen & Gardner, 2010) and its spiritual successor, Extempore (Sorensen, 2011). What makes temporal recursion so interesting in a live coding context is that the function can be modified during run-time so that when it calls itself, it may call a modified version of itself. This allows the programmer to become responsive and dynamic within their own musical programs through what Sorensen and Gardner calls “cyber-physical programming”.

Live coding has been used as a tool for teaching programming within schools in recent years as part of the Sonic-Pi project (Aaron, 2016). Sonic-Pi is a Ruby⁴ based live coding environment created to engage schoolchildren with computing and therefore needed to be simple in design. The result was a constructive teaching mechanism that also doubled as an effective performance tool, which has been used widely in the Algorave community. Sonic-Pi uses the traditional programming concept of loops and “if-statements” to perform music, but with a slight twist; it uses a “live loop” that allows its contents to be edited at run-time, thus allowing it to be live coded. This is a useful way to provide school children with immediate feedback to changes they make in code and the

⁴<https://www.ruby-lang.org/en/>, accessed 06/11/2018

loop functions and has been used effectively in live performances.

```
live_loop :boom do
  with_fx :reverb, room: 1 do
    sample :bd_boom, amp: 10, rate: 1
  end
  sleep 8
end
```

Figure 2.3: Example of a Sonic-Pi live loop

Similar to Sonic-Pi, Earsketch (Xambó, Freeman, Magerko, & Shah, 2016) has been used in schools to help teach children about computer science concepts through music. Earsketch is a web-browser interface that acts as DAW but requires the user to use Python code to organise samples and loops within the piece of music and is (to an extent) live-codable. Earsketch has been designed with collaborative methods of practice in mind but focuses more on collaborative programming as opposed to collaborative performance. Browser-based live coding has been popularised by Roberts and Kuchera-Morin (2012), who introduced the live coding environment, Gibber, which allows the user to access “high-level audio synthesis and sequencing” in their own web browser using JavaScript⁵. Embedding a live coding environment in a web browser relieves the user of having to download and install any prerequisite software, which can often be time-consuming, complicated, and at the expense of hard-disk space. With the software being inherently online it also provides a platform for networked performances between users on the same web page, which is currently facilitated by a Gibber add-on called Gabber (Roberts, Yerkes, Bazo, Wright, & Kuchera-Morin, 2015). The downside to browser-based software, however, is that performances will rely on an internet connection, which is not always available at venues.

Many live coding languages are domain-specific and embedded within existing general-purpose languages, such as TidalCycles within Haskell, Sonic-Pi within Ruby, and Gibber within JavaScript. Another example is FoxDot that is a module for the Python⁶ programming language that comes with its own programmable interface. It focuses on an object oriented programming (OOP) paradigm where agents within the program have states that change over time and can be accessed by other agents in order to build dynamically changing musical systems.

There exist many environments for live coding and more are being developed every year. Those mentioned here are only a handful of the languages that relate to the creation and manipulation of music, but there exist even more for working with graphics and other media. Each language provides a different approach to programming, composition, and performance; each with their own set of strengths and weaknesses. There is no “one size fits all” for the human musicians that use these languages but by having a large variety of them, there is a greater chance that a live coder

⁵<https://www.javascript.com/>, accessed 06/11/2018

⁶<http://python.org/>, accessed 06/11/2018

might find the language that best suits their performative needs.

2.2 Collaboration and Network Music in Live Coding

2.2.1 Network music systems

Many computing terms, such as desktop, window, client and server, are metaphors for real life objects and actions that are designed to make it easier for users to understand the processes taking place on their computer. The term “network” is one such metaphor, which describes the process of sharing information between multiple interconnected computers. We can also think of a group of musicians as a network in that they are all interconnected through the sharing of musical information as they play music together. Jason Freeman closes the loop on this metaphor by saying that “all music is networked. You can think about an Orchestra as a client-server network, where a conductor is ‘serving’ visual information to the ‘client’ musicians, or a peer-to-peer networking model in an improvising Jazz Combo” (Barbosa, 2006, p. 14). With this in mind it feels like it is only a natural progression that, in the internet age, musical networks allow “a group of musicians, located at different physical locations, interact over a network to perform as they would if located in the same room” (Lazzaro & Wawrzynek, 2001). In a technological sense, “network music happens when people make music with computer networks” (Ogborn, 2018) and any live coding performance that involves communication mediated by technology is inherently network music due to its computer-based domain.

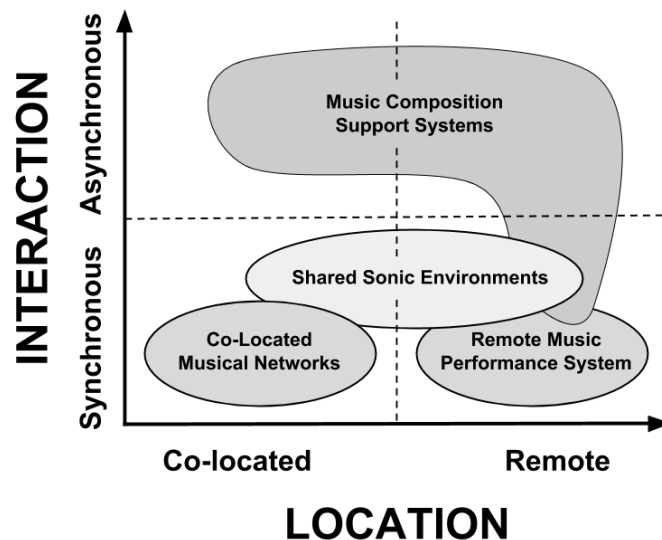


Figure 2.4: Network Music classification graph taken from (Barbosa, 2003).

Network music systems can be categorised along two axes that pertain to the time dimension for interaction, between synchronous and asynchronous, and the spatial location of participants, between co-located and remote (see Figure 2.4). Situated in this 2D plane is a taxonomy of four

categories for network music systems; co-located musical networks, remote music performance systems, shared sonic environments, music composition support systems. For example, early network music performances, such as those by The League of Automatic Composers in the 1970s (Bischoff, Gold, & Horton, 1978), required performers to be co-located due to technological restrictions but modern-day commercially available applications, such as ArtsMesh⁷, now allow musicians to perform together from opposite sides of the world. Music performance is usually a synchronous task but an example of an asynchronous network music system might be a web application for composition that allows multiple editors to work on the same piece from across the internet.

Hugill (2005) defined a taxonomy for network music (referred to as internet music and therefore focuses on network music over wide-area networks) that categorised music systems based on the technique used to generate music as opposed to time and space. They are as follows:

- “Music that Uses the Network to Connect Physical Spaces or Instruments”
- “Music that is Created or Performed in Virtual Environments, or Uses Virtual Instruments”
- “Music that Translates into Sound Aspects of the Network Itself”
- “Music that Uses the Internet to Enable Collaborative Composition or Performance”
- “Music that is Delivered via the Internet, with Varying Degrees of User Interactivity”

It could be argued that, as you move from one end of the location axis to the other, there exists a point at which performers are not physically in the same space together but *feel* that they are together presently through digital representations, such as an avatar or webcam view. This is sometimes referred to as ‘telematic co-location’. We live in an era where we are constantly in communication with one another across the largest computer network in the world; the internet. So ubiquitous is the nature of online sharing via social media and instant messaging (IM) that we forget about the physical distances that separate us and take for granted the fact that we are almost always telematically co-located. Perhaps we should not think of the axis in terms of spatial location but spatial perception and how close we *feel* to our co-performers when using a network music system. Malcolm Arnold said “music is the social act of communication among people, a gesture of friendship, the strongest there is” (Johansson, 2009, p.9) and as inter-personal relationships have moved to a cybernetic plane so too should music performance in a way that retains music’s inherent social nature. As Pauline Oliveros suggests, “as the technology improves exponentially and ubiquitously then eventually there will be no reason not to perform music at a distance” (Oliveros, 2009).

For years network musicians have relied on audio streams to be sent over a network to create music. Even at the speed of light, data takes time to be sent from one machine to another; a delay

⁷<https://www.artsmesh.com/>, accessed 06/11/2018

which is called latency. When this latency is experienced in a perceptible way, it is called network “lag”. This is one of the most common problems encountered in network communication and can occur in a variety of different applications; musical or otherwise. Network lag is most commonly experienced in online gaming and a theoretical in-game interaction provides a useful illustration of the problem and its solution; Two players, A and B, are playing an online shooting game and each person is located in a different part of the world but connected to the same game server. Player A shoots player B but in the time it takes to send the event over the internet, player B may have moved and player A’s shot would now be calculated as inaccurate. How do online games accurately represent real-time events happening in different locations with high levels of latency? The technique for achieving this is known as “lag compensation” (Valve Software, 2017). The server, running its own simulation of the game, keeps a history of all the players’ locations for a small amount of time (around 1 second) and then estimates where a player was, adjusted for lag, when an event occurs. So in this example, the server would move player B back to where they were when player A shot them and both players would be notified of the accurate hit. In a musical sense, it is a bit like two performers sending audio over the internet and both playing slightly ahead of each other before a computer synchronises the two audio streams for an audience. Unfortunately, playing music purposely out of sync in this manner would be a difficult feat to achieve for even highly proficient musicians. This does bare similarities to the piece, *Fields*, by David Bird (2010) in which two performers, separated 150 yards, play a snare drum in time to a personal metronome they can hear in an earpiece in order to accurately compensate for the latency of sound from the other musician and supports the performers in gradually changing tempo independent of one another. Unlike the speed of sound, however, the time it takes to send audio over a network is not always constant and network musicians are often faced with varying latency times, making computer implementations similar to the above more difficult. This variation in latency, known as “jitter”, is often caused by unaccountable forces such as increased network loads or the encryption and decryption of network messages. With larger networks the issue of jitter becomes increasingly noticeable, as each node a network message passes through between its source and destination will add a slight delay. Ogborn (2018) outlines three possible ways of addressing the issue of latency and jitter in network music:

1. Use periodicity in music to synchronise each musician to the previous period of audio received over the network
2. Delay local algorithmic events to synchronise with the reception of events received from other connected performers
3. Avoid metric structures altogether and allow latency and jitter to play a part in the musical performance

Ninjam (Cockos Incorporated, 2018) is one example of software that uses periodicity in music to synchronise musicians over a network. Each connected user sends an audio stream to one another playing at the same tempo then plays along with the stream they then receive; “So when you play through an interval, you’re playing along with the previous interval of everybody else, and they’re playing along with your previous interval”. The problem here is that this is not a natural way of playing music together for many musicians in that they won’t actually be playing with each other at the same time. Ninjam is designed for rehearsal and not live performance as each connected user will be hearing a slightly different combination of audio streams. It could be used to coordinate several performances in separate locations as part of a larger musical project but no two versions of the music would be the same. Ogborn makes use of the second method outlined above in his own ensemble, the very long cat ensemble, using a “zero latency network music configuration” (Ogborn & Mativetsky, 2015). The very long ensemble is made up of one live coder and one tabla player who connect over the internet to play music together. There is a natural delay between the formation and realisation of an idea in live coding so it doesn’t matter if the live coder hears the tabla player at a slight delay as long as the sonic events created by the live coder are then further delayed for both musicians such that they occur at the same time in relation to what the tabla player is playing. This means that “the tabla player simply plays along in sync with what he hears. However, the live coding performer monitors the sounding result of their live coding at a small, calibrated delay equal to approximately the time it would take for network audio [...] to do a complete round trip between the two locations. It is as if the local monitoring of the live coding were to line up in time with itself as transmitted and then transmitted back”. This is an excellent method for synchronisation over a large-scale network but problems arise as more non-algorithmic performers are included in the system. Latency would not be able to be accounted for if there were two tabla players, for example, as they would not be able to play in sync with each other due to network delays. The last method for network music collaboration involves creating non-metrical music or incorporating lag and jitter into the piece of music, such as Atau Tanaka’s *Global String* system, which uses network traffic between connected musicians to generate parameters for audio synthesis (Kim-Boyle, 2009). While embracing the issues inherent to network music as creative constraints, or ignoring them altogether, solves some of the technical problems involved, it does mean that musicians who wish to play together over a network are limited by the type of music they are able to play.

2.2.2 The role of the network in network music

As mentioned previously, a computer network is a technical analogy for a network of social relationships between people but, at the same time, a computer network that facilitates human communication is also a space in which social interaction exists. Music is a social function and,

therefore, network music can be considered a collection of social relationships that are mediated through technology as much as they are by music. Computer networks, and the music systems we use to communicate over them, are public spaces that place constraints on us just as the real world does. “If networks are about relationships between people, then they are also about power, control, and governance. The everyday perception that our lives are evermore influenced by hidden algorithms is only possible because networks connect those algorithms to all of us” (Ogborn, 2018). We are bound by both social conventions established by our culture’s society and the programmatic rules implemented by the software we use to create music. The constraints of any algorithm used in networked software will always govern our actions and affect the way we interact with one another using technology.

Some researchers explore this idea more than others and consider social structures explicitly in developing their network music systems. For example, Fencott and Bryan-Kinns (2010) developed an interface to investigate how privacy and awareness within the system affected group interaction and Knotts (2016) implemented a voting system to instil a level of democracy in a large telematic ensemble. Even without addressing social relationships in the design of a network music system, they will, by their very nature, affect the way users behave as part of a social group. So how do we consider network music system in the context of human (acoustic) music performance? Are they a multi-user instrument? Or perhaps, they are performance pieces in their own right? With the notion of a computer network being a social space to communicate in, one could think of improvised network music as something close to John Cage’s *musicircus* 1967, in which any number of performers are invited into a space to “perform simultaneously anything or in any way they desire”⁸. Cage was one of the first to separate form and material in music composition and in *musicircus* the only form is the simultaneous music-making in a single space, with the material being created by the performers in any way they please. In this regard, the development of technology that allows multiple people to share a network space to make music together could be considered a loose form of composition. However, composer Max Neuhaus regards his network music projects “not as a self-contained musical work in itself but as a collective instrument or musical architecture through which participants develop relationships through musical dialog” (Kim-Boyle, 2009). Perhaps network music systems are more about nurturing social relationships in a way that give the users more power and control over the music, as opposed to a more authoritative musical work. Let us then examine the network music systems that exist for live coding and how they are used to facilitate musical and social communication in ensemble performance.

⁸https://johncage.org/pp/John-Cage-Work-Detail.cfm?work_ID=273, accessed: 22/11/2018

2.2.3 Network music systems for collaborative live coding

Live coding is a technology-based performance practice that can be extended through software to create networked music systems for ensemble performance. This can be done by adapting the existing functionality of a live coding environment or through the use of an external application designed for enabling musical communication between live coders. Shared timing information is an important aspect of traditional ensemble performance (Janata & Grafton, 2003) and when live coding metric-based music, performers must also share timing information across their computers using some application or protocol. The TidalCycles language (McLean & Wiggins, 2010) allows users to tightly synchronise their performance environments as long as their computer clocks are synchronised using a common network time protocol (NTP) (Mills et al., 1985), such as Network Time Protocol daemon (ntpd) or Precision Time Protocol daemon (ptpd)⁹. The SuperCollider language also enables performers to synchronise their environments' clocks from within the language itself but requires additional program libraries, known as 'quarks', to be installed or developed. For example, the Birmingham Ensemble for Electroacoustic Research use the 'Listening Clocks' quark (Wilson, Lorway, Coull, Vasilakos, & Moyers, 2014) and popular live coding ensemble, Benoit and the Mandelbrots¹⁰, have developed their own clock synchronisation library, 'BenoitLib'¹¹, to manage the sharing of timing information across a network.

One of the characteristics of live coding that make it so unique is its direct manipulation of musical material through computer code. It is unsurprising, then, that practitioners have been exploring methods for collaboratively working with code in a variety of ways. As with coordinating timing information over a network, distributing and manipulating code across multiple performers requires some protocol or application for facilitating the process. This might be a distributed memory across a network that acts as a virtual "bulletin-board" that users can post and retrieve data from as is the case with the live coding environment, Impromptu (Sorensen, 2010). Users coding within Impromptu can write a tuple of data to the memory space that contains the item of data and a corresponding signature, such as <"tempo". 120>, which can then be read from the memory space by requesting a tuple with a signature that matches "tempo". The live coding duo, aa-cell, have also used the Impromptu environment to incorporate data sharing into their collaborative practice by using its Inter Process Communication mechanism, which allows them to define functions and variables on the other's computer directly (Brown & Sorensen, 2007). In contrast to this method, the PowerBooks Unplugged ensemble shares their SuperCollider code with each other, as opposed to data, to be edited and re-purposed however other members of the ensemble feels (Rohrhuber et al., 2007). With this system, performers sit at various points in the

⁹https://tidalcycles.org/howtos.html#multi_laptop, accessed: 21/11/18

¹⁰<http://www.the-mandelbrots.de/>, accessed: 05/02/2017

¹¹<https://github.com/cappelnord/BenoitLib>, accessed: 05/02/2017

room and write small portions of code for generating sound, called “codelets”, that are shared with the rest of the ensemble using an interface similar to an IM application. They then edit the incoming codelets and use them to generate slight variations of their co-performers’ audio, creating a musical experience that changes depending on the listeners location within the room. This performance system has been further developed and made publicly available as a library called ‘The Republic’ (de Campo, 2014). Similar to The Republic is LOLC, which is a “textual performance environment” that aims to facilitate methods of practice common to both improvisation and composition, with a focus on conversational communication (Freeman & Van Troyer, 2011). It is a platform for sharing shorthand musical patterns, which are then played or transformed and re-shared by other performers. It cites avant-garde ensemble improvisation pieces such as John Zorn’s *Cobra* (Zorn, 1987), which only defines rules for communication and not music, as a consideration for its design. This is particularly interesting as it postulates the idea that network performance systems could be regarded as compositions themselves, similar to the comparison drawn between network music systems and John Cage’s *Musicircus* in Section 2.2.2.

Code sharing and collaboration can also be achieved without IM-style applications, such as through the browser-based system Extramuros (Ogborn, Tsabary, Jarvis, Cárdenas, & McLean, 2015), which allocates each connected performer a small text box on a web page into which they can input code. These text boxes are visible to any other connected performer, who are also free to make edits. Extramuros is a “language-neutral” application and can be used with any language that allows commands to be “piped” into it, which improves accessibility for a wider range of live coding practitioners. This does require the languages to be configured to take text as an input, but popular environments, TidalCycles and SuperCollider, already work with the platform. By incorporating all of the performers’ code into one web page, it can be displayed to the audience using only one projector. This solves many issues that prevent live code ensembles from projecting their code, which is often “nontrivial to implement with five or more performers” (Wilson et al., 2014). In contrast to The Republic, code evaluated by one user in Extramuros is evaluated for every other connected user simultaneously.

With browser-based software comes web applications and communication over the internet. Live coding language, Gibber (Roberts & Kuchera-Morin, 2012), is run in the browser and can be used collaboratively with other users on a variety of levels using a library extension called *Gabber* (Roberts et al., 2015). This is a combination of a chat-room interface and shared text buffers similar to the Extramuros platform. Unlike Extramuros, text buffers are not all publicly displayed and are only visible by clicking on a user’s name within the chat-room. Users can write code in a “personal editor”, which is executed only on their own machine, and in a “shared editor” that other connected users can contribute to. Gabber uses two modes of what it calls “state sharing” for collaborative live coding; local (the default) and remote mode. In the local mode, users are

expected to be co-located and generate their own unique audio on their local machine but can also execute code on their co-performers' machines. Using certain keyboard shortcuts, users can broadcast code from within their personal editor to be executed on all connected machines or they can evaluate code in a shared editor that is run on only for the users connected to that shared editor. In contrast, the remote mode for Gabber assumes that all users are separated geographically and all code is executed on every performers' machines.

Collaborative live coding over the internet is not restricted to just browser-based environments and most modern programming languages provide an interface for connecting to other applications over the internet, known as web services. SuperCopair (de Carvalho Junior, Lee, & Essl, 2015) is a package developed in Coffee Script for the Atom.io editor¹² that allows users to collaborate over the internet with the SuperCollider live coding environment. It combines functionality from two existing packages, atom-supercollider and atom-pair, to utilise cloud service, Pusher¹³, to push code execution actions across the internet to other users. SuperCopair provides users with higher levels of control over how code is executed than other collaborative live coding interfaces; users can choose whether code is only run locally or broadcast to other connected users, and they can also decide if code received from other users is run immediately or if they would like to be asked permission beforehand.

Much of the research in collaborative live coding has focused on temporal synchronisation, code sharing, and instant messaging, but little on the transfer of audio streams directly over a network. By working only with code as the medium for ensemble communication, live coders give up the opportunity to collaborate with other technology-based practitioners, especially in telematic music performance. To enable the collaboration of performers (live coders and otherwise) over large geographical areas, Knotts (2016) developed several systems, such as "Union" and "Flock", which utilise the transfer of audio data, as opposed to code, over a network. For examples, the "Union" system uses a Music Information Retrieval (MIR) library in SuperCollider to analyse incoming audio data from connected performers and mixes them together based on spectral similarity. One performer is present in the performance space and acts as the curator of this process by ensuring that the collated music is coherent, and then broadcasts the mixed audio out via the internet to be listened to by online audiences and also the performers themselves. This is particularly interesting as performers are subject to 5-20 second delay between sending their audio stream and hearing it mixed together in the final output. High latency in data transfer is one of the biggest problems in using audio data directly in network music and can make it very hard for performers spread over large areas to collaborate closely with their musical material. Using an artificial intelligence to mix different streams together at the time of receiving data may help to overcome this challenge aesthetically, but brings into question the performer-composer relationship with regards to the

¹²<https://atom.io/>, accessed: 23/11/2018

¹³<https://pusher.com>, accessed: 23/11/2018

computer.

Live coding collaboration is not always technologically homogeneous; that is, not all performers use the same software to create music together. When this is the case, performers will often need to coordinate timing information through an extra piece of software or adapt their own performance environment. This is well exemplified by Algorave pioneers, Slub, whose members use different software but do “make a network on stage, but this is only to create a shared clock so that [they] may coordinate tempo changes, and share the same down beat” (McLean, 2015). When this is the case, an extra layer of technology is required to coordinate timing across a network. This highlights the difficulty in realising one of the most fundamental musical facets of ensemble performance in live coding and network music as a whole; playing in time. In an effort to make this process as simple as possible, Ogborn (2012) developed the tool, EspGrid, which enables clock synchronisation for any live coding language that can send and receive OSC messages. This allows multiple different live coding languages to synchronise their clocks and even enables live coding languages to synchronise with other music-making software. The EspGrid software runs in the background and finds other EspGrid instances running on any computer connected to the same network, which then agree on a metric structure to use. Live coding environments can then query the local EspGrid for the tempo and beat values for scheduling their own musical events. Developers can create “helper objects” that are built into the various pieces of music software to make it easier to communicate with EspGrid, which already exist in popular software such as ChucK, Max, and SuperCollider and can be easily developed for other technologies. This helps address one of the most difficult issues with collaborating using the technologies discussed above; the pre-requisite that everyone involved uses the same language. Traditional ensembles typically use multiple different instruments as they all speak the same language; music. With live coding, however, performers will often need to use the same language to collaborate using a specific interface, such as Extramuros or Gabber. The browser-based interface, ‘Estuary’ (Ogborn, Beverley, del Angel, Tsabary, & McLean, 2017), which was originally designed to only utilise TidalCycles, now enables several different live coding languages to be used simultaneously (Del Angel, Teixido, Ocelotl, Cotrina, & Ogborn, 2019). Unfortunately, the languages accessible to Estuary are only based in Haskell so languages such as FoxDot and Sonic-Pi cannot currently be used with it. However, this is a strong indication that multi-lingual live coding is not only possible, but could become standard practice in the future.

Not only do live coders collaborate with one another using different software, but there are also several instances of live coders performing with non-live coders in contexts both mediated and unmediated by technology. An example of the latter would be the piece *Fermata* for bass clarinet and Threnoscope (Furniss, 2016). The Threnoscope is a live coding system developed by Magnusson (2013) that is designed for 8 channel speaker systems and focuses on the use of microtonal drones. Magnusson considers it a musical work in its own right that affords high levels of improvisation but

has been combined with other instruments to create new pieces, such as *Fermata*. In this piece, collaboration occurs only at the listening level and is not facilitated by any other technology, such as the sharing of code or audio data. The Threnoscope has also been combined with instruments that have been augmented by technology, such as the resonating cello (Eldridge & Kiefer, 2017), as part of the practices of the *Braindead Ensemble* (Polimeneas-Liontiris, Eldridge, Kiefer, & Magnusson, 2018). Audio signals generated by the Threnoscope are played through speakers and captured by transducers attached to the bodies of acoustic instruments that resonate and create a new signal that is captured by the instrument's pickups. This act of live coding collaboration is occurring at the hardware level and creates a "acoustic network" as opposed to a network created with computers. The live coding and piano duo Off<>ZZ¹⁴ have collaborated for several years and have explored various methods for performing together. This often involves the augmented piano, known as the Codeklavier (Veinberg & Noriega, 2018), that allows the performer to use the piano as an interface for live coding. In a recent performance at the International Conference on Live Coding 2019 in Madrid, Off<>ZZ used the Codeklavier to perform lambda calculus on input data from the live coder and then route the output back into live coded musical algorithms (Veinberg & Noriega, 2019). Both hardware and software technologies have been implemented in this instance, which allows data to be manipulated and shared between live coder and non-live coder, which facilitates a deeper level of inter-performer collaboration. The live coding and electronic drum kit act, Canute, share data in a similar fashion. The drummer "produces probability distributions of hits on his drum kit, visualising them and sending them" to the live coder to be transformed using TidalCycles (McLean, 2015). Data is how live coding represents music and if traditional instruments can be augmented to create and manipulate it, then there are many exciting possibilities for heterogeneous collaboration in the field of live coding.

As well as augmenting physical instruments and reducing network latency, several practitioners have developed specific interfaces to enable live coders and non-live coders to collaborate effectively. Lee and Essl (2013) developed a system in which a performer uses a tablet to interact with a tone matrix (pitch in the y-axis and time in the x-axis) and a live coder can modify its contents during the performance. This might involve changing the number of columns and rows or altering the timbre of the voice being triggered by the performer. This creates a very interesting relationship between performers that draws parallels with the 'leader-follower' roles found in many other styles of music making. The tablet performer (follower) is wholly constrained by the features of the interface presented to them by the live coder (leader). That being said, the live coder does not contribute any musical material (unless through another application) to the performance but defines the rules for what material is possible to create. To give non-live coders more consistency in a collaborative interface, Sarwate, Rose, Freeman, and Armitage (2018) developed four prototypes for facilitating

¹⁴<https://www.keyboardsunite.com/offzz/>, accessed: 07/05/2019

collaboration between live coders and non-live coders. The first was a program that allowed a non-coder to create a melody using a Novation Launchpad¹⁵, which would then be transformed by the live coder in a call and response format. However, the complex mechanism for triggering the transformations and lack of visual feedback for the non-coder made it very difficult to use. The second prototype gave the non-coder more visual feedback and utilised a persistent loop structure that the non-coder could update via the Novation Launchpad and the live coder could update using code. While the live coder could also transform the whole sequence the non-coder was not able to undo these change, which did not satisfy practical requirements. However, Sarwate et al. felt this prototype did improve on the first. The third prototype explored signal processing by allowing the live coder to map custom curve functions to parameters in the effect chain that the non-coder’s output signal (via an electric guitar) was going through. The non-coder was not able to relate the continuous data shown in the interface to the discrete musical data they were used to and struggled to generate musical material. The last prototype was a physics based interface in which balls bounced off the walls in a square and triggered notes when lines drawn between the walls were crossed. Non-coders could adjust the trajectory of the balls using a MIDI controller, which could also be used to manipulate global effects such as filters and delays, and the live coder could adjust the location of the lines (that triggered notes when crossed) and their musical mappings. The physics based style of making music led to some challenges in interacting effectively due to its unpredictability and complexity, such as difficulty creating rhythms by changing the location of lines based on the trajectory of a ball when it has been altered. Based on this study, Sarwate et al. suggested that, when designing interfaces that enable collaboration between live coders and non-live coders, the “[m]usical state should be represented visualized in a mutually modifiable interface with musical representations intuitively familiar to the performers”.

2.2.4 Futures of live coding collaboration

What does the future hold for collaboration in live coding? It is a difficult question to answer but there are some who have made predictions about what multi-person opportunities may be available in the future.

An area of further great potential is collaborative or competitive coding. Performers can pass code to each other to modify, allowing a very abstract sense of musical transformation, and even work in a Chinese whisper style remix circle. Games might be set up where coders have a fixed time limit to complete some goal with a restricted set of tools. One could even imagine a live coding version of the rap ‘dis’ battle where coders compete to aurally insult one another, or a hacker style ‘root war’ in which they subvert each other’s computer systems. (Collins et al., 2003, p. 322)

¹⁵<https://novationmusic.com/launch/launchpad>, accessed: 07/05/2019

Ensemble performance is likely going to be part of the future of live coding, but will performances be of a collaborative or competitive nature? Competition provides a platform for novel improvisations but performance systems specifically designed for this style of musical interaction are, perhaps, not generalisable to a broader range of musical styles. This PhD thesis aims to develop live coding interfaces for facilitating collaborative creativity that can be employed in a wide range of musical and performance styles. In it I present a total of three live coding interfaces and the extension of a live coding language for ensemble performance. The first interface, Troop, is a collaborative text editor for live coding that enables live coders to work together. Where Extramuros separates users' code into text boxes, Troop combines concurrent, real-time coding into a single text buffer using colour as a means for identifying which live coder has written what. Troop makes use of the FoxDot live coding language, whose functionality has been extended as part of this research. Syntax has been added that allows users to share information between music-generating agents in a simple and robust way, enabling a more collaborative style of coding. Where many collaborate environments allow data to be shared between instances, such as Impromptu, FoxDot embeds this process within in its core data-model, allowing live coders to create musical relationships in the code that adapt to changes made by co-performers.

Another two graphical user interfaces are also presented in this thesis. The first is CodeBank, which inspired by version control tools used for collaborative software engineering projects. In an attempt to present that process within a microcosm in musical performance, CodeBank provides each performer with a private workspace that they are free to test and try out ideas before sharing their code publicly with an audience and their co-performers. This interface shares many similarities to the Republic performance system, using the distribution of small sections of code known as "codelets" as the main medium for collaboration. In contrast to the Republic's decentralised audio generation, CodeBank utilises a public repository where all codelets are evaluated to be heard by the audience. Finally, the Polyglot interface explores cross-language collaboration by extending the concurrent text-editing ability of Troop to combine multiple live coding interpreters into a single interface. It uses the decentralised network time-keeping tool EspGrid to synchronise multiple languages into a single, coherent performance system. Polyglot shares many of the same ideals as the Estuary interface; to allow any group of live coders to collaborate together. However, it also shares the same distinctions that the Troop interface has with Estuary's predecessor, Extramuros, which is that the editor does not distinguish users by text box but allows them to edit code together.

3. Method

3.1 Introduction

This research project is a practice-based investigation into ensemble performance in live coding, which aims to address the research questions outlined below. This chapter discusses the motivations for pursuing this area of research and outlines the methods and processes used to explore collaboration in the practice of live coding.

3.2 Rationale for Research

My first introduction to live coding was in 2014 and, since then, I have been developing my practice, as well as my own performance language (see Chapter 4 for an in-depth discussion). I performed primarily as a solo improviser of electronic dance music but it became an increasingly lonely experience as time went on. I attempted to remedy this by collaborating with an electronic musician / drummer but found that there were limitations to what we could achieve together when combining our two separate performance systems. One of the most challenging aspects of the collaboration was that we were unable to see the other's screens or share data between systems and the rehearsals that we did have were unrewarding. This is not always the case for computer musicians collaborating with heterogenous software systems, see Slub (McLean, 2015) for example, but even when I was personally able to perform with other live coders I felt that there needed to be something in place to facilitate a more satisfying and creative collaboration. As described by Derek Bailey in the quote below, finding fruitful collaborations became a priority for me in order to progress as an practitioner of improvised music.

For most people improvisation, although a vehicle for self-expression, is about playing with other people and some of the greatest opportunities provided by free improvisation are in the exploration of relationships between players. (Bailey, 1992, p. 105)

Consequently, the main motivation for this PhD is developing my own creative practice and successfully moving it from a solo performance context into a group one. Where many practice-based computer music PhD commentaries present software as part of a series of musical works, such as Armitage (2017) and Knotts (2018) for example, I aim to develop more generalisable platforms for facilitating improvisation while simultaneously exploring how these platforms affect group interaction. As discussed in the previous chapter, there already exist several methods for

collaborative live coding but few that allow me to work with, or develop, my own performance software. This research project allows me to further my practice as an improvising live coder through developing new interfaces for collaboration using my own live coding language.

What makes a good interface for collaboration live coding? Can existing methods for collaborative live coding be improved upon? These are some of the questions sit at the crux of this thesis. From first hand experience as a live coding practitioner I have found that the majority of live coding performances, especially of those who are inexperienced, are enacted as a solo performance. There are a number of explanations for this: the existing methods for collaboration are not effective enough at facilitating ensemble interaction, software is too complex to set up for new performers, or there may be lack of locally available collaborators. Perhaps it just that most live coders are content to perform on their own due to the breadth of control they are afforded by their software. The solo performer has seemingly become the de facto in electronic music as technology has developed to give much more control to a single person over a live performance. Emmerson states that in the field of electronic music “the sound producing power of a solo performer has become polyphonic – capable of creating many layers of the musical stream. Thus the move to groups of performers in this field has been tentative and solo artists remain a majority” (Emmerson, 2007, p. 114). He goes on to note that IDM labels such as Warp and Rephlex are “dominated by solo artists”, further indicating that the world of electronic music tends to be a solitary one. This is not to say that electronic music is performed solely by individuals; many of the most prominent acts in recent years, such as Autechre and Orbital, are made up of multiple performers. Research has shown that playing music and singing together is beneficial to one’s well-being (Clift et al., 2010; MacDonald, 2013), but this benefit is perhaps often overlooked in electronic music. Developing effective interfaces for collaborative live coding could allow electronic musicians to come together and enjoy the social benefits of group music making through live coding. Another downside of the abundance of solo live coding performances is that it perhaps gives the impression to onlookers that “this is the way things are done” and that an authentic live coding performance is only achieved through performing on your own and this only serves to perpetuate the commonplace nature of solo live coding.

Live coders usually project their screens so audience members can experience the creative processes that take place during performance; but do interfaces for ensemble live coding also reveal the processes that occur in inter-performer communication? Several methods for collaborating involve using multiple screens, some of which are not projected, and this surely conceals the communicative actions that might take place within the code. Even when every ensemble members’ code is visible, are the responses to their co-performer’s actions even clear to an audience? There are many shared aspects between improvised jazz performance and live coding (Burland & McLean, 2016) and part of the excitement for audiences at a jazz gig is attributed to “seeing the interaction

of the players” (Burland & Pitts, 2012), so why shouldn’t that be the same for live coding? Few existing collaborative interfaces compile all of the code being written by a group into one place, the notable exception being Extramuros (Ogborn et al., 2015), and this obfuscation of ensemble communication may discourage live coders from performing in ensembles. As a practice embedded in the culture of improvisation it feels like a wasted opportunity that not more live coders are performing together.

The overarching goal of this research is to develop interfaces that are more effective for facilitating collaboration in ensemble live coding. In order to do this I need to examine what “more effective” means in this context. Successful facilitation of a collaborative performance will have an effect on the outcomes of the musical activity, but how can that effect be measured? Is it the production of a higher aesthetic quality of music, or is it the cultivation of synergy within an ensemble? Or perhaps an interface that shares with the audience the communicative signalling that takes place across the ensemble is best? Is it possible to achieve all of the above or must some aspect be sacrificed to realise the others? For all musicians creating better music is in their mind when they pursue a new creative endeavour so I will not be addressing this explicitly in my research questions, although the quality of the produced music will be a consideration when evaluating an interface. This presents two research questions that, when answered, will provide insightful knowledge into collaborative live coding and the role that software can play in it;

1. How can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?
2. How are collaborative interfaces used to reveal the creative processes at play in ensemble live coding performance?

By addressing these questions through the development of collaborative interfaces for ensemble live coding, I will explore the themes of group creativity, joint action, and ensemble communication as well as crossing the disciplinary boundaries of music and computer science.

3.3 Methodology

3.3.1 Research in the wild

This research aims to produce software that will explore various aspects of ensemble performance, computer programming, and any other areas of research that fall between these fields. Much of the work will be grounded in practice-led research but at the crux of this work is the *design* of interfaces that enable live coding in a collaborative context. With regards to the research questions outlined above, interfaces will be designed to facilitate ensemble live coding and reveal creative

processes. The design of anything is rarely perfect in its first iteration and is often improved over time through use in the world by those it was designed for:

Much good design evolves: the design is tested, problem areas are discovered and modified, and then it is continually retested and remodified until time, energy, and resources run out.

(Norman, 1998, p. 142)

A methodology that uses design as the focal point of practical research is known as “participatory design” (Spinuzzi, 2005). It frames the process of design as the research itself; the production of artefacts, systems, and practical or tacit knowledge. It draws on a range of research methods to “iteratively construct the emerging design, which itself simultaneously constitutes and elicits the research result as co-interpreted by the designer-researchers and the participants who will use the design”. This philosophy forms the foundation on which the software development will be carried out as part of this research. Interfaces will be developed for collaborative live coding using an evolutionary and user-centred design approach. Each interface that is created will undergo several “design phases” in which it will be created and modified, used and tested, and its success or failure reflected upon and discussed. The outcomes of these reflections will feed into the following development phase and this process will continue until a satisfactory result is achieved. Broadly speaking there are two types of methodological approach for evaluating software for computer supported cooperative work; ethnographic studies that aim to “characterise the interaction and group processes” in naturalistic observations and laboratory studies that take place in controlled environments (Fencott, 2012). Xambó, Laney, Dobbyn, and Jorda (2011) suggest that the evaluation of musical interfaces should take place in contexts that are as close to “real” as possible, which deters me from pursuing a laboratory-based methodology for this project. Ethnographic studies are useful for gaining deep understanding of users’ behaviour but are often difficult to perform as they require access to a group of musicians for a sustained period of time and that group must be comfortable using novel or uncommon interfaces. New technology takes time to learn and observations can only realistically take place over longer periods than those afforded to lab-based studies. In participatory design the research methods should be “shot through the entire research project; the goal is not just to empirically understand the activity, but also to simultaneously envision, shape, and transcend it in ways the workers find to be positive” (Spinuzzi, 2005). Longer term observational studies also allow musicians who already play together and have developed chemistry to use new technologies in realistic rehearsal and performance contexts and ‘forget’ that they are in a study and demonstrate more authentic behaviours. These methods have been used to study electro-acoustic musicians presented with prototyped software (Merritt, Kow, Ng, McGee, & Wyse, 2010) and suggests that similar approach would be an appropriate methodology in this instance.

For these reasons I am choosing to pursue an ethnographic study; observing a group of performers in both rehearsal and performance settings over large periods of time coupled with reflexive interviews in order to assess the quality of the interfaces and explore how they impact group creativity and communication. Evaluation of the software from real-world users will be intrinsic to the design process itself, and is a key aspect of participatory design research (Bødker, Grønbaek, & Kyng, 1995) and has been used to develop a range of digital musical technologies Geiger et al. (2008); Landry and Jeon (2017).

From an artistic perspective, a longer length study is also beneficial with regards to developing a portfolio of musical works. As a practice based heavily in improvisation, much of the success of any performances will depend on good ensemble chemistry and confidence using the interfaces. Without being able to use the interfaces for longer periods of time these would be difficult to develop. An ethnographic approach to this project also allows for the documentation and analysis of how the ensemble chemistry develops over time. Gifford et al. (2017) proposed several considerations that should be made when evaluating interfaces for musical improvisation; namely the “trust, risk and responsibility” of the interface, ability to achieve “flow”, and the sense of “immediacy” that the interface gives its users. In their paper the “general approaches to evaluating the quality of a particular interface, and the success of a given improvisation” are tailored to improvisation as a solo performer as opposed to an ensemble, but lay solid foundations for assessing the success of digital interfaces for improvised music making and will be considered in the context of collaboration in place of individual and computer creativity. Gifford et al. go on to say that the best way to evaluate an interface is by just using it and reflecting on how good the experience feels. These guidelines will be used as part of the evaluation of the interfaces, which will take place in the form of personal reflection and group interviews with users.

Integral to the software development process is continual evaluation and feedback into the design through live performances, or “research in the wild” (Benford et al., 2013). There are several reasons for pursuing this method; it puts technology into the hands of the artists who can find new and unintended uses, it engages the public and enriches cultural life within society, and, most importantly, it provides an opportunity to test the software in the context for which it was designed:

The public deployment of artworks offers a test-bed for putting emerging technologies into the hands of users in a ‘realistic’ situation, meaning a situation in which the technology needs to be made to work and is treated in some sense a professional product (Benford et al., 2013)

It should be stated that in their paper, Benford et al. frame “research in the wild” as a performance-led methodology, as opposed to a practice-based one and there is an important dis-

inction between these two methods and should be clarified. Where Benford et al. are concerned with the refinement of a singular artwork through real world instantiations of it, practice-based research examines all aspects of artistic work that lead to the production of an artefact, such as a performance. In the context of this research project the software design, alongside musical performance, is considered key to the practice and, consequently, the research. Software development is not just the means to an ends here, but also an artistic process that will produce contributions to the understanding of collaborative live coding as well as craft knowledge through reflection and study. The “research in the wild” methodology may be proposed with respect to developing artworks as opposed to supportive tools for creating art, but much of the process can be implemented in the same way. It consists of three inter-related activities that all inform each other in some way; practice, studies, and theory (their relationships are outlined in Figure 3.1). Practice, as the name suggests, is the development of works led by the artist. In this research it consists of both the development of software for collaborative live coding and the creative performances that utilise them. The second activity is studying the artwork, which, in this instance, will consist of evaluating the user experience of the software in practice and considering its impact on improvisational creativity. Lastly, there is theorising. This is the generation of models or frameworks, or even just identifying phenomena that take place when experiencing the artwork. Where the studies explain *what* is happening within the practice, theories bring together results of the studies to explain to *why* they are happening.

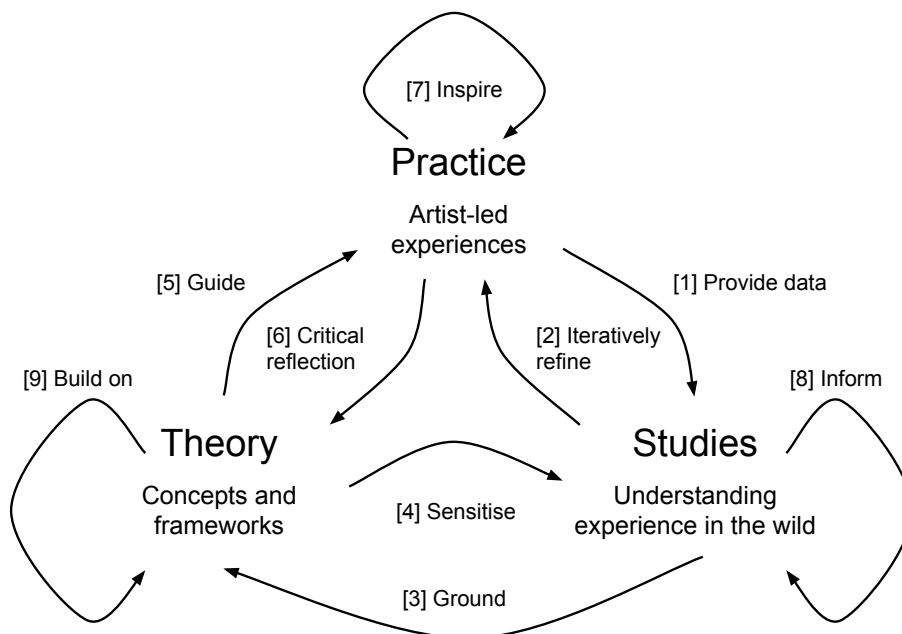


Figure 3.1: Overview of performance-led research in the wild, adapted from (Benford et al., 2013)

Where time and sample size allows, quantitative data will also be collected from users through the use of anonymous online surveys. Combining quantitative and qualitative data collection is

known as “across-method” methodological triangulation (Bekhet & Zauszniewski, 2012) and is used to enhance understanding and increase the validity of results. One of the weaknesses of using an ethnographic methodology is the limited sample size used for observations, which will usually only consist of one or two small groups of participants. Triangulating the methodologies can enrich the study and provide another perspective when evaluating the interfaces. Adapted from the “Questionnaire for User Interface Satisfaction” (Chin, Diehl, & Norman, 1988), the survey will ask participants to rate various aspects of an interface and also provide written answers where appropriate. Similar questionnaires have been used to evaluate live coding interfaces, such as *ixi-lang* (Magnusson, 2011b), and will help provide useful information into the general usability of the developed systems.

3.3.2 Participants

In this section I will introduce myself as a practitioner-researcher and outline my role within the ensemble-based practice that takes place over the course of this PhD; I will not only be designing the live coding interfaces, but also incorporating them into my own practice and reflecting on my experience. Autoethnography is the process of interpreting one’s actions in a as-close-to objective manner as possible, positioning the researcher as “the object of inquiry” in an ethnographic study (Crawford, 1996). It is often used as a technique for understanding one’s own practice Bartleet (2009); Spry (2016) but it has also been used in HCI to inform requirements analysis and system design (Cunningham & Jones, 2005). Of course, this research will also involve other musicians as it is a study of collaboration and ensemble performance and similar self reflexive methods have been used in ensemble contexts under the name of “informed observation” (McCaleb, 2011). This work combines both ethnographic and autoethnographic approaches in its pursuit of examining the creative processes and inter-personal communication that occurs in musical practice and feeds the results back into the design-testing-refinement cycle of developing software for collaborative live coding. In his book, *The Design of Everyday Things*, Norman advocates that design should be “based on the needs and interests of the user” but suggests that the designer is not a good representation of the end-user:

Designers are not typical users. They become so expert in using the object they have designed that they cannot believe that anyone else might have problems; only interaction and testing with actual users throughout the design process can forestall that.

(Norman, 1998, p. 151)

Being both the developer and a practitioner does mean that there is an inherent bias in the design of the software I use and this will also be the case when developing collaborative interfaces as part of this research. However, using the software with other live coders as part of an ensemble

will allow me to collect rich data from the user feedback and observations and incorporate it into subsequent software design phases. The practice-based work will be carried out with a newly formed ensemble made up of musicians with varying levels of experience in music, programming, and live coding as a whole. This will help collect data that is more representative of the average user, compared to an expert, and put the user at the centre of the design process. This originally consisted of myself and computer musicians, Lucy Cheesman and Laurie Johnson; all of whom are based in the wider Yorkshire area and prompted us to name our collaboration The Yorkshire Programming Ensemble or TYPE for short. Lucy is an accomplished live coder who performs using the Tidal Cycles language under the moniker of “Heavy Lifting” and Laurie, while an experienced performer and programmer in his own right, had not done any live coding prior to the start of this project. The ensemble’s line-up was expanded to include a novice live coder, Innocent Granger, in December 2018. One of the difficulties of performing long-term observational studies is getting consistent access to the performer’s schedule but by being part of the ensemble itself I am guaranteeing complete insight into the rehearsal and performance processes of the group.

3.3.3 Considerations

As mentioned in Section 3.2, new or inexperienced live coders tend to perform in a solo capacity. I see this as problematic for several reasons; for a first-time performer, getting up on stage and overcoming nerves is a difficult task, made even more challenging when having to deal with these anxieties on your own. The reason for the prevalence of solo performance in live coding is, in part, due to the technical challenge of setting up existing collaborative systems, which requires a certain level of expertise. This is just one of several barriers to entry in live coding; a preexisting level of technical knowledge is required. Even installing programs can be a difficult task for some and, with almost all live coding software being open source, there is often limited documentation. If you do manage to install your live coding language of choice, you then need to learn how to use it. Live coding is not an intuitive way of making music; your first time will probably involve an empty text editor, with no buttons or dials that indicate how to make sound. This, too, requires reading through minimal documentation, as well as watching tutorial videos, and a lot of trial and error. This process is especially difficult if you have never done any computer programming before as it means you also have to learn a lot of fundamental concepts, such as syntax, data structures, and functions. Now that you’ve installed all the software you need and mastered the language, you want to play music with other people. However, you now need to install more programs and libraries, which usually require some technical understanding of computer networking, to do so. At each point in this journey there are points of failure that might stop the progression of a potential live coder and live coding ensemble; software may not install correctly, documentation might not exist, and you may just not have the technical knowledge to implement a particular live coding

system.

3.4 Chapter structure

Each interface developed as part of this research will be presented primarily within its own chapter, which will discuss its motivation, development, and use in practice. The creation of a completely original idea is a rare thing indeed and, so, the design of any interface will, of course, take inspiration from existing studies into collaborative music making. Each chapter will outline the motivation for developing a specific interface and draw upon the advantages and disadvantages of technologies that have addressed similar problems previously.

The main body of work in each chapter will relate to the practice-based research and involve “documenting and describing the work” extensively (Benford et al., 2013). The development process will be iterative and take place over several “phases” and each phase will contain documentation on the design and development of the interface as well as a written description of any live performances. All performances will be recorded using a video camera recorder, or through screen recordings where appropriate, and provided on an accompanying SD Card. Following the documentation will be a discussion of the performance on the impact the interface had on its success or failure.

The final section of each chapter will be its conclusion, in which the interface, and its effect on performance and ensemble communication, will be evaluated. The framework for evaluation is taken from (Gifford et al., 2017), which provides the guidelines for the personal reflections and participant interviews that are presented in this section. Gifford et al. posit that the best way to evaluate an interface is by just using it and reflecting on how good the experience feels, and this notion sits at the heart of this research.

At the core of the research questions outlined in this PhD, there is one fundamental question posed with regards to collaborative live coding; how can it be improved? But what does this mean? Improving ensemble performance applies both to the performers and to the audience; making the communication of ideas clearer for both those using the systems and also those experiencing them as listeners and viewers. Live coding is an audiovisual experience and its use of code projection can sometimes leave audience members feeling distracted or even confused. Increasing the number of performers could potentially open up the possibility of alienating audiences by further obscuring the creative processes through the use of more screens or complicated performance systems. The research aims to address these issues by developing user-centred interfaces that provide performers with simple means of collaboration and an audience-engaging system image, while also improving the stagecraft and creative output of live coding ensembles.

4. Foundation Work

4.1 Introduction

My interest in live coding began about 18 months prior to the start of this PhD when studying for my masters in Computer Music at the University of Leeds. I fell in love with the idea of composition as performance but found that there were limitations with the currently available software. The two most popular live coding environments at this point seemed to be TidalCycles and SuperCollider but neither satisfied the requirements I had for improvising with programming languages. SuperCollider’s strength is in its flexible sound-synthesis capabilities but the verbosity of its syntax means that even simple sequencing of a few notes requires significant amounts of typing. I wanted the low-level control over the sound but felt that the delay between forming an idea and implementing it as code was too great and this meant that the development of the music was too slow. On the other hand, TidalCycles is a much more terse language that focuses on the quick manipulation of musical patterns. In 2015, though, it was not possible to trigger or manipulate software synthesisers, which was something I wanted to utilise in my own practice. In response to this I began developing the live coding environment, FoxDot, to allow me to control software synthesisers built with SuperCollider and also sample playback in a succinct and dynamic manner. I have continued to develop FoxDot and have since performed with it across the U.K. and abroad. FoxDot features heavily as part of the practice undertaken during this PhD project and this chapter describes its key features and syntax in order to better inform the reader going forward in this document.

4.2 FoxDot

FoxDot is a live coding environment based in the Python programming language and focuses on readability and OOP in the manipulation of musical patterns (Kirkbride, 2016). It is very similar to the TidalCycles language developed by Alex McLean (McLean & Wiggins, 2010) but allows users to program musical patterns of synthesised sounds that are stored in SuperCollider as opposed to just audio samples, which was not possible in TidalCycles when I began developing FoxDot. Shortly after this, however, TidalCycles added this functionality through the SuperCollider “SuperDirt” quark¹. Even though both languages share many similarities, FoxDot differentiates itself by incorporating more elements for traditional western music, such as scales and meter, that do

¹<https://github.com/yaxu/SuperDirt>, accessed: 06/11/2018

not feature as prominently in TidalCycles. This was a conscious decision to not only appeal to users coming from backgrounds rooted in more traditional music theory but also make any musical knowledge gained through its use transferable to other musical practices. Similarly, the choice of Python as the host language makes FoxDot accessible to a wider audience; Python is one of the most popular programming languages in the world² and its simple and clean syntax also makes it great for newer programmers as well.

FoxDot operates primarily as a middleware between the user and the SuperCollider audio engine and acts as both a replacement for SuperCollider’s own language, SCLang, and a framework for music making that provides users with a library of pre-programmed synths and pattern functions. This setup allows users to spend less time typing and worrying about *how* to do something and more time being creative. I have been constantly supported in my development of FoxDot by the creator of TidalCycles, Alex McLean, and it has since taken on a life of its own and has developed a user base from all over the world including Europe, South America, and Asia.

4.2.1 Player objects

FoxDot reserves all lower-case, two-character variable names, e.g. `aa`, `p1`, and `bd`, to be used as “players objects”. These are the objects that play audio based on instructions given to them. The first instruction given is telling it what instrument, known as a “SynthDef”, to play. Users assign a SynthDef (short for synth definition) to a player object by using two ‘greater than’ signs, called the ‘double arrow’ in FoxDot, followed by the name of the SynthDef and a pair of brackets. For example, to assign the “pluck” synth to the player object `p1` you would use the code `p1 >> pluck()`. This would start playing a single note on repeat until the player object is told to stop using `p1.stop()`.

```
p1 >> pads([0, 1, 2, 3], dur=8, oct=(4, 5))
```

Figure 4.1: Example of typical FoxDot code.

More instructions can be given to player objects by supplying arguments in the brackets following the name of the SynthDef. The first argument is the pitch of the notes, which are written as degrees of the current musical scale. Other arguments need to be specified with a keyword but can be done so in any order. Keyword arguments usually use a shorthand version of their description e.g. `dur` to specify duration and `oct` for octave. Each argument can be a single value, a list of values that are used in sequence, or a group of values that specify multiple sound events to be played simultaneously. For example, the snippet of code in Figure 4.1 shows a player object using a

²<http://pypl.github.io/PYPL.html>, accessed: 13-02-16

SynthDef called “pads” to play the first four notes of the scale in both the fourth and fifth octaves for eight beats at a time.

```
# Summing the product of two lists using FoxDot
p1 >> pads([0, 1, 2, 3]) + [0, 0, 4]

# Summing the product of two lists using Python's builtin library
p2 >> pads([sum(x) for x in itertools.product([0, 1, 2, 3], [0, 0, 4])])
```

Figure 4.2: Comparison of FoxDot code with Python’s standard library.

A list of values can also be added to the player object to create variations in a sequence in a very simple way, whereas achieving the same outcome using Python’s standard library of functions is more difficult and verbose. For example, the code in Figure 4.2 shows two player object playing the exact same sequence of notes, the first four notes of the scale with every third note shifted up four steps, but executed in different ways; the first is simply using the ‘plus’ operator on the player object to add the values together and the second is using Python’s `itertools` module to calculate the Cartesian product of the lists and summing the values.

```
# Using groups of values within a sequence
p1 >> pluck([(0, 2, 4), 1, 2, (3, 5, 7, 9)])

# Adding a tuple of values to create a chord sequence
p2 >> pluck([0, 1, 2, 3]) + (0, 2, 4)

# Using a group to play notes in multiple octaves
p3 >> pluck([0, 1, 2, 3], oct=(4, 5, 6))
```

Figure 4.3: FoxDot code for playing multiple notes simultaneously.

Multiple notes can be played simultaneously from within the same player object by grouping values into a tuple data structure using round brackets. This can be done either within a sequence or added to the player object to create a chord sequence, as shown in Figure 4.3. These can also be used in other attributes to play notes across multiple octaves, in separate channels, and to combine multiple levels of effects concurrently.

FoxDot can also be used play back audio samples using a special SynthDef called `play`. Instead of taking a list of numbers as its first argument it takes a string of characters where each character is mapped to a different sound, similar to how samples are triggered in `ixi-lang`. Encoding the information in a string allows it to be parsed for information about how the sequence should be played back. The parser looks for different types of brackets, which manipulate the sequence in various ways. For example, putting two or more characters in round brackets will alternate which sample played on each loop through the sequence and wrapping characters in square brackets will


```

# Simple sequence of samples
p1 >> play("x-o-")

# Using brackets to alternate samples used
p2 >> play("(x-)(-x)o-")

# Playing a triplet of hi-hat samples on the 4th step
p3 >> play("x-o[---]")

# Using curly braces to select a sample at random
p4 >> play("x-{o-x}-")

```

Figure 4.4: Example FoxDot code using the `play` SynthDef.

play them successively in the duration of one step, as shown in Figure 4.4.

```

# Reversing a sequence every 3 beats
p1 >> play("x-o-").every(3, "reverse")

# Plays a note 4 steps up on the offbeat
p2 >> pluck([0, 1, 2, 3]).sometimes("offadd", 4)

# Randomise the order of the duration sequence every 3 beats
p3 >> pluck(dur=[0.75, 0.75, 0.5]).every(3, "dur.shuffle")

```

Figure 4.5: Example FoxDot code using the `every` method.

FoxDot also allows users to transform any sequence used by a player object by calling functions at regular intervals using the `every` method. This tells the player object to transform a sequence using a specific function every `n` number of beats. By default the player object will perform the transformation on the pitch or sample sequence, but another attribute can be specified by name with the function, as shown in Figure 4.5. The frequency of these transformations can also be indeterminate; using the method `often`, `sometimes`, and `rarely` will call the function at random intervals within high to low frequency ranges.

4.2.2 Patterns

A `Pattern` in FoxDot is a container-type object, similar to Python lists, but much more versatile for managing and transforming sequences. Python lists are not very flexible when it comes to transformations and, as demonstrated in Figure 4.2, even summing the product of two lists is an overcomplicated task. The `Pattern` data-type gives users the ability to combine multiple sequences using traditional mathematical operations and allows custom behaviour to be added to simple data structures. A Python list is transformed into a `Pattern` by simply prepending it with an upper-case 'P', as shown in Figure 4.6. The `Pattern` object then handles any transformation logic applied to it, such as basic arithmetic or other methods.

```

# Creating a simple Pattern object
my_pattern = P[0, 1, 2, 3]

# Repeat each element 3 times
print(my_pattern.stutter(3))
-> P[0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3]

# Reverse the entire contents
print(my_pattern.stutter(3).reverse())
-> P[3, 3, 3, 2, 2, 2, 1, 1, 1, 0, 0, 0]

# Multiply by another Pattern object
print(my_pattern.stutter(3).reverse() * P[2, 1])
-> P[6, 3, 6, 2, 4, 2, 2, 1, 2, 0, 0, 0]

```

Figure 4.6: FoxDot `Pattern` transformations and output.

Python lists only have a small number of methods for transforming their contents. They can insert or remove single values, extend the list with the contents of another, or reverse the order. This is not a satisfactory amount of control for the manipulation of sequences for music. Another downside is that these methods change the list *in place*, which means they do not return a new version of the list, but transform the list and return nothing. This makes composing multiple transformations for Python lists difficult as it is not possible to immediately call a second method. Custom functions can be written to combine multiple transformations but, as shown in Figure 4.7, the order, and subject, of the transformations is not always clear. Readability and control flow is better when using methods attached to a single object, which is another reason why the `Pattern` object was developed.

FoxDot also has a collection of functions that generate useful `Pattern` sequences, such as a linear series. Using Python’s “slicing” syntax, users can create a large sequence using very few characters; instead of explicitly writing every value, users can specify the start, end, and step size of the series separated by a colon. For example, a user could generate a `Pattern` of the odd numbers between 1 and 10 by typing `P[1:10:2]`. Multiple linear series can be combined into a single `Pattern` by separating the specifications with a comma, as shown in Figure 4.8.

```

# Using Pattern method to perform transformations
P[0, 1, 2, 3].stutter(3).reverse() * P[2, 1]

# Using functions to perform transformations on a list
multiply(stutter(reverse([0, 1, 2, 3]), 3), [2, 1])

```

Figure 4.7: Comparison of transforming FoxDot `Pattern` objects and Python lists.

Another set of functions for generating `Pattern` objects, known as ‘named functions’, are functions that take single value inputs and output a new `Pattern`. These can vary from simple

```

# Combining series in a single Pattern
print(P[1:10:2, 5:0:-1])
-> P[1, 3, 5, 7, 9, 5, 4, 3, 2, 1]

# Generating durations using Euclidean rhythms
print(PDur(3, 8))
-> P[0.75, 0.75, 0.5]

# Create a Pattern of length x, whose last values is y
print(PStep(5, 4, 0))
-> P[0, 0, 0, 0, 4]

# Example of looping named Pattern functions
print(PStep([5, 3], 4, 0))
-> P[0, 0, 0, 0, 4, 0, 0, 4]

```

Figure 4.8: Example `Pattern` functions used to generate sequences.

functions, such as `PStep`, which takes 3 values, `x`, `y`, and `z`, and returns a sequence of `x` length that ends in `y` with every other value set to `z`, to more complex functions, such as `PDur`, that calculates Euclidean rhythms (Toussaint et al., 2005). Named functions can be ‘looped’ by supplying a list of values as an input, which will extend the `Pattern` with the output of the function for each input value in the list, as shown in Figure 4.8.

4.2.3 Time-dependant variables

Music, for many people, is the change in sound over time and representing it as a series of events in sequence may not always be the most effective way. In `FoxDot`, for example, a user may have a chord sequence in which each chord is played for 8 beats each. Using a duration of 1 beat per note, it would be relatively simple to work out that each chord needs to be played 8 times in a row, and could easily be implemented by ‘stuttering’ a `Pattern` 8 times. Changing the duration of the player object, though, would require the user to update the number of times the `Pattern` would need to be ‘stuttered’ in order to hold the correct pitch values for 8 beats at a time. If the duration was changed to 1/4 beats then the user would need to stutter the `Pattern` 32 times. Things get more complicated when a sequence of different durations is used; if the duration was changed to something like $[3/4, 3/4, 1/2]$ then the user now has a more difficult calculation to do in order to stutter their chord sequence. This will only add to their cognitive load, break their flow, and slow down the performance. What happens if the duration is a random values? How would one organise their sequence to play each chord for 8 beats? The answer to this was to introduce a data-type called a time-dependant variable, or “`TimeVar`”, for short, that holds a value for a duration of time as opposed to being a step-by-step sequence.

A `TimeVar` is instantiated using the keyword `var` and takes a list of values and durations as arguments, as shown in Figure 4.9. Any transformation made to the value, such as addition or

```

# A TimeVar is used to hold pitch values for 8 beats
p1 >> pads(var([0, 1, 2, 3], dur=8), dur=1/4)

# A TimeVar is used to create a chord sequence
p2 >> pads(var([0, 1, 2, 3], dur=8) + (0, 2, 4), dur=1/4)

```

Figure 4.9: Example FoxDot code using a TimeVar.

multiplication, returns a new TimeVar that contains a relationship to the original such that when the value changes, so too does the new, transformed value. For example, a TimeVar is created that hold the values 0 and 1 for 4 beats each and a second TimeVar is created by multiplying the first by 2. When the first TimeVar holds the value 0, the second will also hold the value of 0. When the original TimeVar holds the value of 1 after 4 beats have elapsed, the second TimeVar then holds the value of 2. This can be useful for creating chord sequences and shared variables that are used by multiple player objects that change their value at the time, as demonstrated in Figure 4.10.

```

# A TimeVar is created
chords = var([0, 3, 0, 4], [16, 8, 4, 4])

# It is used to create a chord sequence backing
p1 >> pads(chords + (0, 2, 4), dur=chords.dur)

# And a melody that revolves around the tonic of the chord
p2 >> pluck(chords + [3, 2, 0], dur=1/2)

```

Figure 4.10: Example FoxDot code using a TimeVar shared between two player objects.

4.3 Considerations

This is a practice-led PhD and FoxDot forms the basis of my current musical practice. Throughout this research I will be exploring how user interfaces can be vehicles for expressive ensemble live coding performance with FoxDot at its core. The motivation for this research stems from the desire to play music with others in a way that goes beyond the levels of interaction that currently exist within the landscape of live coding. I am aiming to shift my own practice from a solitary venture to a social one and bring FoxDot with me on this journey. By developing my practice-led research around FoxDot I am definitely placing a constraint on myself, but FoxDot is at the heart of my practice and my practice is at the heart of this research. This chapter should not only provide the reader with a better understanding of the syntax that will be discussed frequently throughout this document, but also provide them with the insight into *why* this research features FoxDot so prominently.

5. Troop: An Interface for Real-Time Collaborative Live Coding

5.1 Introduction

This chapter introduces the collaborative live coding interface, Troop, that allows users to work on the same piece of code together simultaneously. Troop gives audiences insight into the performer’s interactions, not just their musical decisions, and offers performers a holistic view of the textual material enabling direct interaction with one another.

5.2 Motivation

On a typical laptop performance, Emmerson notes that creative actions are usually heard and not seen:

At its most paradoxical the ‘laptop performer’ may move little and think a lot: the clues of *will*, *choice*, and *intention* will be inferred from the sounding flow or through apparent responses to the sounds of other performers (Emmerson, 2007, p. 112)

By contrast, the practice of live coding often strives to give audiences insight into performances through the projection of code, which is a key part of the TOPLAP manifesto: “Give us access to the performer’s mind, to the whole human instrument. Obscurantism is dangerous. Show us your screens” (TOPLAP, n.d.). The projection of the performer’s screen is seen as fundamental to live coding performance but audiences reception can be divisive (Burland & McLean, 2016). Code and music are intrinsically linked during performance and this is understood by the audience who often expect to be able to follow musical events and relate what is on the screen to the consequent sonic experience (Magnusson, 2011a). The downside of this is that some audience members feel that watching the screen “pulls the focus away from the human performers and the listening” (Burland & McLean, 2016, p. 10). Burland and McLean suggest that by projecting their work performers allow their code to become a representation of themselves but one could argue that this is only accessible to a minority who are able to decipher the code’s meaning. In comparison to other styles of performance live coding can be relatively static; performers may move in rhythm with their music but the constraint of using a computer keyboard to compose/perform in real-time does not afford very expressive movement. As a consequence the code becomes the outlet for self

expression and, without it, the audience will rarely gain insight into the mind of the performer. Research has shown that being able to see a performer in close proximity improves the experience of live music (Burland & Pitts, 2012) and live coders often rely on the projection of their code to develop a level of intimacy with the audience.

As the number of live coders performing together increases, so too does the number of screens required to project all of the ensemble’s code. This is a well documented issue and some live coding ensembles, such as the Birmingham Ensemble for Electroacoustic Research (BEER), are not always able to project their code in its entirety:

BEER has experimented with the common live coding practice of projecting code [...] although we have not done this consistently. One reason is simply that this raises additional technical demands, which can be difficult to meet in some situations. The second is that, although we appreciate the way in which code projection can enhance an audience’s experience by making clearer connections between performer activity and the resulting sound, it is nontrivial to implement with five or more performers. (Wilson et al., 2014, p.55)

The relationship between the audience members and the projected code is not universal and even those who appreciate that its role is integral to the performance style of live coding believe there is room for improvement. One participant from Burland and McLean’s study stated “I really enjoy seeing the projected code. I still think the community has a long way to go in terms of stagecraft while preserving the legibility of code” (p. 10). Zmólnig (2016) suggests that a potential solution “is to provide additional information on the screen that is not a presentation of the source code itself, but instead some kind of visualisation of the running program”. Zmólnig is referring to the computational processes involved in a live coded performance but perhaps the idea of the “running program” could also be interpreted as the cognitive processes that occur during performance, such as those involved with musical creativity, but the argument could be made that code’s projection in itself already addresses this.

Live coding shares many of the same characteristics that make live jazz performances so interesting in that each performance is usually improvised and therefore unique (Magnusson, 2014). Music is created in the moment and the aspect of risk and uncertainty adds excitement to the event as it does in jazz (Burland & Pitts, 2012). Jazz audiences want “to be close to the musicians, see them interact with each other and see them play as clearly as they could hear them” (Brand, Sloboda, Saul, & Hathaway, 2012, p. 9), which suggests that the *creative process* accounts for as much of the appeal of improvised jazz as the music itself. The concept of sharing with the audience the creative processes that emerge from the dynamic interaction between performers is fundamental to the design of Troop.

In a similar manner to Google Docs (Google Inc., 2017) Troop uses a shared text buffer, which enables concurrent word processing over the internet. A live coding editor that enables this style of collaboration was proposed as an addition to the Impromptu live coding environment (Brown & Sorensen, 2007) but was never realised. Troop aims to make this a reality and allow live coders to collaborate directly on the same material such that they can see, as well as be part of, the live composition. Working within the same document live coders not only share their immediate textual material with one another but also their cognitive workspace; traces of each contributor's thought patterns are left in the document as each keystroke updates the code, visible for all to see. This on-screen interaction makes the collaborative process transparent and accessible to the audience and allows the performers to share their musical ideas with one another in real-time. Each performer is allocated a different coloured font to help distinguish their individual contributions. By comparison, contributions in Google Docs can only be separated when viewing a document's version history and not in real-time.

Without being able to see co-performers' code with the use of a shared text buffer, live coders often rely on performers listening to one another's contributions, which creates a latency between the instantiation of a musical idea and a corresponding co-performer's reaction. It could be argued that this stifles the interactive processes that appeal to audiences when watching improvised music. In addition to this, the audience's attention can be divided when performers' code is projected across multiple screens. If the audience feels that the projection of just one screen "pulls the focus away from the human performers and the listening" then dividing their attention between multiple screens could only have a detrimental effect on an audience's response. In some cases, a lack of projection equipment may mean not every performer's screen is visible but in either scenario the process of creative collaboration is obfuscated by the separation of the performers' expressive representations in their code.

As discussed in section 2.2.3 there are several existing programs that facilitate collaborative live coding such as Extramuros (Ogborn et al., 2015), The Republic (de Campo, 2014), and LOLC (Freeman & Van Troyer, 2011). Many, if not all, of these do not address collaboration at the textual level, which is the primary medium in which the practice of live coding takes place. Extramuros, for example, allocates each connected performer a small text box on a web page into which they can each write code. This allows them to create their own code and request and modify other performer's code within the same window but it does not necessarily encourage users to develop music together using the same textual material. It does, however, reduce the number of screens necessary to project during the performance, which could improve audience experiences.

Gabber (Roberts et al., 2015), the network performance extension to the JavaScript based language, Gibber, does allow users to interact with each others code directly within the same text buffer. This allows for only one screen to be projected but all performers' work to be displayed.

Originally Gabber used “a single, shared code editor” but it was found to be problematic “as program fragments frequently shifted in screen location during performances and disturbed the coding of performers”. It also did not show the cursor locations of performers, which no doubt made it hard to see who was editing what part of code at any one time. Gabber has since moved to a more distributed model in which users can request and edit another user’s code in a similar manner to Extramuros. The single, shared text buffer model may have proved problematic in this instance but it has seen much mainstream success, most notably in Google Docs (Google Inc., 2017) and SubEthaEdit (TheCodingMonkeys GmbH, 2014), and has prompted the development of the present collaborative editor, Troop.

5.3 Phase 1: Inital Implementation

The purpose of the Troop software is to allow multiple live coders to collaborate within the same document in such a way that both performers and audience members alike will be able to identify the changes made by the different contributors. It also reduces the technical complexity of setting up multiple screen projections by collating all of the coding activity into one window, which can be viewed from a single projector. This section outlines the initial steps taken to achieve this and briefly describes the network architecture and communication protocol used throughout this project.

5.3.1 Development

During this phase Troop was designed to interface solely with the Python-based live coding language, FoxDot (Kirkbride, 2016). Python is useful for rapid prototyping and comes with a built-in package for designing graphical user interfaces (GUIs) called Tkinter. This, along with its ability to easily import FoxDot into its environment, was the reason Python was chosen as the language for Troop’s development. The philosophy behind Troop is that all performers should seemingly share the same text buffer and contribute to its contents simultaneously. To allow each performer to differentiate their own contributions from others each connected performer is given a different coloured label that contains their name. This label’s position within the input text box is mapped to the location of the respective performer’s text cursor, as shown in Figure 5.1.

Over the course of a performance, the text buffer naturally would begin to fill up as more code is written by performers. With multiple collaborators it would not always be possible to separate the individual contributions and it becomes unclear as to whom has written what; even to the performers themselves. This problem was identified by Xambó et al. (2016), stating there is a challenge in “identifying how to know what is each other’s code, as well as how to know who has modified it”. To help combat this problem and differentiate each performers’ contributions each

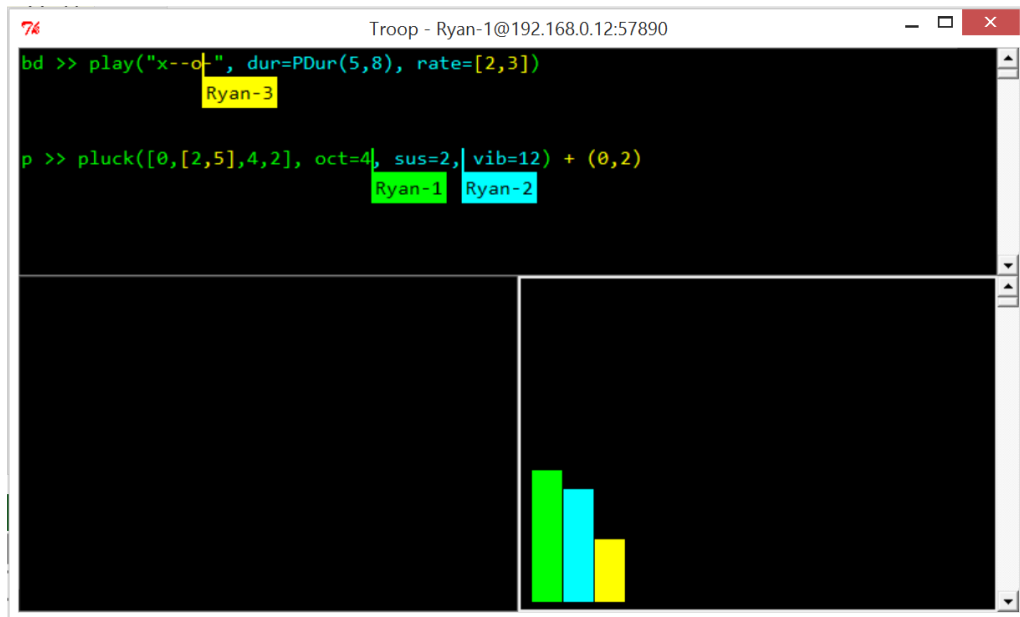


Figure 5.1: Early version of the Troop interface.

connected performer is allocated a different colour for text (see Figure 5.1) and highlighting. By doing this performers can leave traces of their own coloured code throughout the communal text buffer. This is an example of one of Norman’s user-centered design principles; using technology “to make visible what would otherwise be invisible, thus improving feedback and the ability to keep control” (Norman, 1998, p. 192). Editing a block of another performer’s code interweaves both their font colours and also their thought processes, creating a lasting visual testament to a collaborative process; or at least until someone else makes their own edit. The colour of the text entered by each performer matches the colour of their marker to retain a consistency in each performers’ identity. It is customary in live coding for evaluated code to be temporarily highlighted and by doing this in separate colours it allows both audience and performers to identify the source of that action.

Performers’ contributions are also measured in terms of quantity of characters present. In the bottom right corner of Figure 5.1 there is a bar-chart that displays the proportion of code contributed by each performer. As a collaborative performance tool it is particularly interesting to keep track of this information and could be used as a creative constraint for performances in the future. Finally, the background colour for the text area was changed and line numbers were added, which helps users identify locations of specific code when communicating about the code itself, shown in Figure 5.2.

To edit code together, the text in Troop’s buffer needs to be replicated across all performers’ computers such that when one user enters a new character, it appears in the same position for all other users also. The simplest way to implement this is to use a centralised synchronous networked performance model, similar to that of LOLC, where each performer uses a client application that

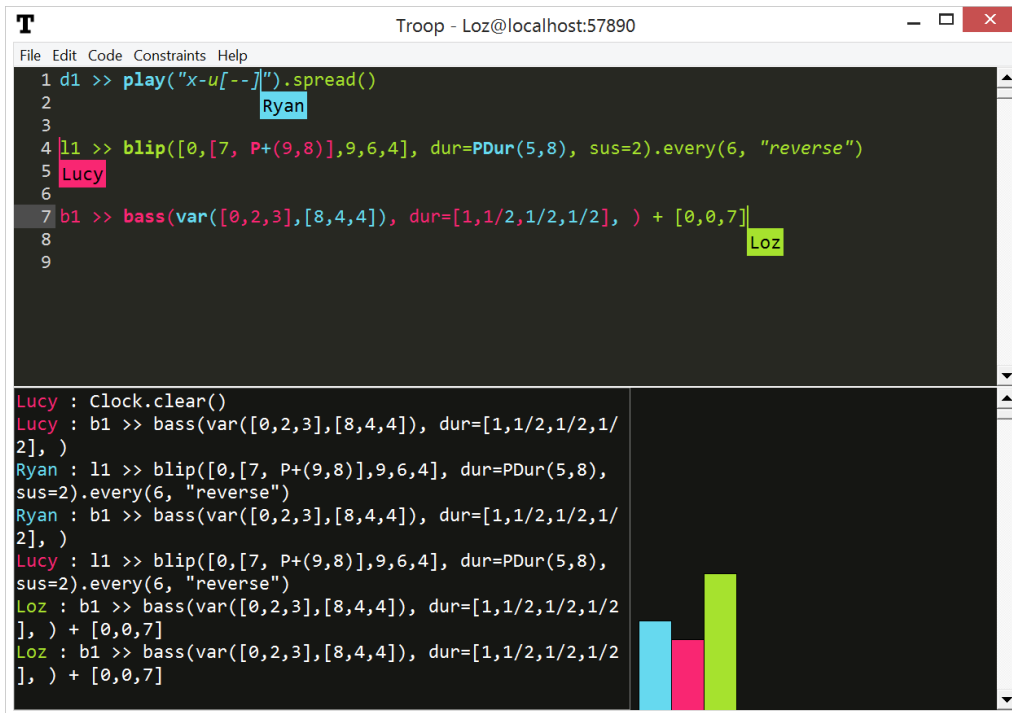


Figure 5.2: Final design of the Troop interface.

connects to a server, which manages all communication between the clients (see Figure 5.3). When a user presses a key, the client application sends a message to the server with this information and stores it in a queue data structure. The server then forwards this message to all connected clients in the order that the queue received them. This is also the case for mouse-clicks and direction-button presses on the keyboard, which update the location of a user’s text cursor. This ensures that all client applications receive the same data in the same order. The use of decentralised network topologies is also common in networked live coding performance (Lee & Essl, 2014) but storing the correct order of messages is critically important and this is best handled by a dedicated application running on the centralised server. Messages are transferred using TCP/IP to ensure they are all sent over the network.

A disadvantage to storing messages in a queue on the server is that there is sometimes a noticeable delay between a user pressing a key and the corresponding letter appearing on that user’s screen. Once a key is pressed, a client must wait for the information to be sent to the server and then back again before it appears in the text editor. During periods of high activity the message queue becomes “backed up”, which increases the delay. If the server application is running on the local machine then the latency is negligible but connecting over the internet to a server running in a data centre in London can be problematic; on average it takes a few milliseconds for a character to appear but the latency between key-press and the character being displayed could be up to several seconds depending on the level of activity and speed of the internet connection. While this does keep the text synchronised across all clients, it was not an ideal user experience during testing.

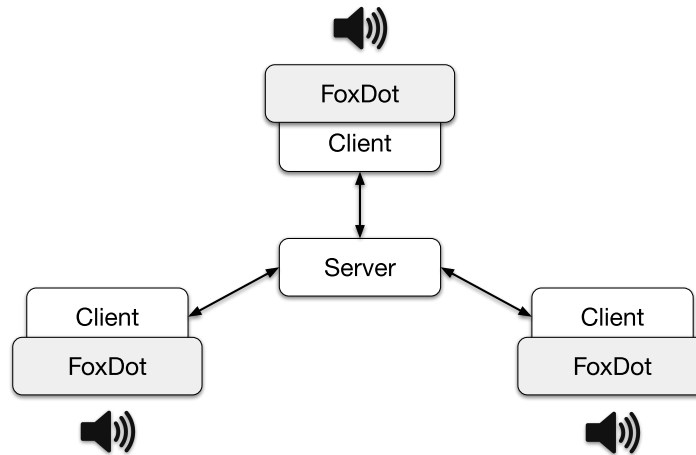


Figure 5.3: Troop's Network diagram.

The proposed remedy was to process local key presses immediately on the client instead of waiting for them to be sent to the server. This made the system much more responsive but was prone to having clashes in the order of the text between the clients, such as those seen in Figure 5.4. This issue mainly arose when two users were editing the same line, caused by keystrokes being processed immediately on a local machine not being processed in the same order as the queue residing on the server, thus giving each connected user a different body of text. This issue was exacerbated when users were deleting multiple characters, especially over multiple lines, causing several lines of text to merged into one jumbled line. As the number of connected clients increased so too would the chance of this error occurring. To combat this, the original text synchronisation algorithm was re-implemented for users editing the same line, such that keystroke messages were sent to the server and processed in the same order. Keystroke messages were still processed immediately in all other cases, improving the responsiveness of the program while still minimising as many clashes as possible.

```

d1 >> play("-(- [==])- ([xxxx] )o")
#l1.stop()p1)*2, sus=2, chop=4, lpf=linvar([0,2000],16)).follow(b1) + [0,4,2]
lucy

```

Figure 5.4: Example of scrambled text in Troop.

The nature of Troop's concurrent text editing means that code in each client's text buffer is identical so there is no need to use a shared name-space, such as those used in (Lee & Essl, 2014; Sorensen, 2010), as the data is reproduced on each connected machine. An advantageous consequence of this is that Troop need only send raw text across the network and no data needs to be serialised in order for the program to be replicated on multiple machines. All code is evaluated on every connected client's machine so that users connected over a wide area network, such as the

internet, can all hear the same music.

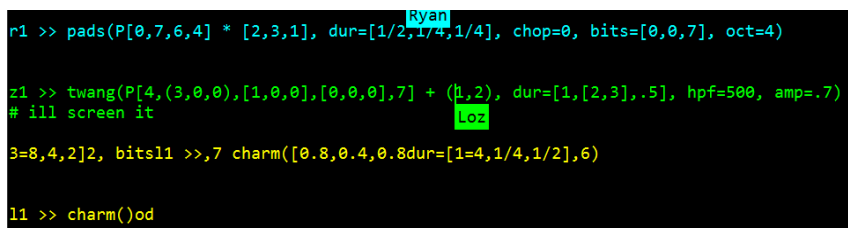
5.3.2 Practice

Initial rehearsal sessions, various locations - 09/04/17

Video recording: `ch5_1a-Rehearsal-09_04_17.mpg`.

The first few instances using Troop with members of TYPE were run as online workshop sessions as both other members of the group were new to the FoxDot live coding language. This meant that much of the text was used as direct communication as opposed to musical code. Embedding the instructional conversation within working code examples that could then be directly manipulated by other participants was very useful and helped express ideas and concepts that otherwise might have been hard to grasp. Due to the discursive nature of these sessions, most of the text was used to ask questions and describe functions or syntax rather than create working, musical code. This created a very saturated screen as lines of code were separated by blocks of comments and discussion. It was difficult to understand what was happening outside of your own code and was described as “chaotic”. Lucy suggested we discuss strategies for the rehearsal for future sessions.

The music from these sessions, such as the one recorded and included in this thesis, was often simple and rigidly metrical. Learning to use Troop also required Laurie and Lucy to learn how to use FoxDot and their inexperience was reflected in the music. Although simple there were, however, some interesting textural elements at times as we began to experiment with the array of effects available in the software. There were also some issues with text consistency that occurred when multiple users were editing lines close together. The yellow text at the bottom of the editor in Figure 5.5 is in a nonsensical order even though it was written by only one user. It was unclear what caused this error but the likely cause was that another user deleted text spanning multiple lines while the yellow text was being entered but before the location of cursor was updated. This error only occurred on a small number of occasions but was significant enough to force everyone to stop any sound, delete all the text, and start over.

A screenshot of a Troop client interface showing a code editor with several lines of text. The text is color-coded: 'r1' is red, 'z1' is green, and 'l1' is yellow. The text is as follows:

```
r1 >> pads(P[0,7,6,4] * [2,3,1], dur=[1/2,1/4,1/4], chop=0, bits=[0,0,7], oct=4)

z1 >> twang(P[4,(3,0,0),[1,0,0],[0,0,0],7] + (l,2), dur=[1,[2,3],.5], hpf=500, amp=.7)
# ill screen it

3=8,4,2]2, bitsl1 >>,7 charm([0.8,0.4,0.8dur=[1=4,1/4,1/2],6)

l1 >> charm()od
```

Figure 5.5: Screenshot of inconsistent text contents across Troop clients.

Leeds Algorave, Open Data Institute, Leeds - 28/04/17

Video recording: `ch5_1b-Leeds_Algorave-28_04_17.mp4`.

See Appendix A.1 for performance description.

TYPE's inaugural public performance took place in Leeds at the Open Data Institute (ODI) at an Algorave organised as part of Leeds Digital Festival¹. This was an event that I organised alongside Alex McLean and Joanne Armitage alongside the ODI team funded by WRoCAH as part of their knowledge exchange program. The event featured 10 artists live coding music and visuals and also hosted workshops the following day. Performance set-up differed to the majority of our rehearsals in that we were co-located and audio only needed be produced from a single laptop. My laptop ran the server application and generated audio and Lucy and Laurie connected to the server using the client application over the ODI Wi-Fi.



Figure 5.6: The Yorkshire Programming Ensemble. Photo by Aaron Ratcliffe.

5.3.3 Evaluation and outcomes

One of the most surprising things about these early rehearsals and performances was how we achieved a balance between each performer's varied musical influences. Lucy is a sound artist who often works without tempo or metre, Laurie performs in a “doom jazz” band and makes/produces drone music, and I come from a background of indie-rock and melodic techno. We all had experience with improvisation but in very different ways. The Algorave itself helped provide an aesthetic direction for this performance; music for people to dance to. This helped pin each of our eclectic musical backgrounds to a common thread; repetition. Much of the music produced was

¹<https://leedsdigitalfestival.org/>, accessed 06/11/2018

centered around repeated rhythmic and melodic sequences with slight variations but both Lucy's and Laurie's affinity for timbre based music did shine through at points. Towards the end of the set in particular were some rougher textures created on the fly that lent themselves well to the club night vibe. Much of group improvisation stems from the reactions to a co-performer's actions and the dynamism with which a performance evolves. The TidalCycles language was developed to be particularly terse and allow performers to react quicker to other musicians' actions for just this reason (McLean, 2015). As a group that has only started learning how to use both Troop and FoxDot a few weeks prior, the ability to react in a dynamic way had not yet come and there were long periods of simple, repeated structures. Similarly, the group had only been playing together for a short period of time and many live coding groups, such as ALGOBBZ who were also performing that night, develop their chemistry over a number of years. At this stage in the project it could just be that the quality of music was a reflection of our lack of chemistry as opposed to the limitations of the software.

One of the immediate benefits of the Troop interface compared with other methods for live coding collaboration for the ease of set-up. Audio is not distributed across multiple laptops in Troop, which means only one laptop is required to be connected to the speaker system. Because of this, the instances of FoxDot running on each machine do not need to be tightly synchronised, which further reduces the complexity of set up for performance. Simplifying the set-up for collaborative performance helps reduce the barrier to entry for newer live coders who may be put off more complex systems if they don't have the necessary technical knowledge to operate them.

However, there were still several issues with the system with regards to its functionality. On several occasions, in both rehearsal and performance, text was became scrambled and inconsistent across multiple clients. For Troop to give performers the experience of editing the same body of code it needs to be identical for every connected user. The algorithm currently implemented for achieving this is unsatisfactory and should be improved. During the Leeds Algorave performance these errors did act as an impetus for creative decisions by forcing us to start from scratch, but this, at its root, is an unwanted behaviour for the system. The idea of being forced to start over is not new in live coding; the ixi-lang environment contains a "suicide" function that, when executed, will randomly crash the system at the start of a bar (Hutchins, 2015). Perhaps this practice could be utilised as a tactic for future TYPE performances.

An unexpected outcome from this initial phase of development was the effectiveness of Troop as a teaching tool. Early rehearsal sessions were run as informal workshops for teaching the basics of the FoxDot language, with many of them taking place from several different locations over the internet. Users could write questions about specific parts of the code and I could either answer directly or even write new code to demonstrate certain features. This opens up exciting possibilities for hosting live coding workshops over even larger geographical areas for people who live in parts

of the world without access to a local live coding community. As a pedagogical tool itself, Troop allows teachers work directly with their student’s material and correct any minor syntax errors while allowing them to continue to practice without being interrupted by errors.

This phase of development has shown that Troop has the potential to be a powerful tool for collaboration but the high occurrence of errors and inconsistencies in the text leaves room for improvement. One way to ensure text is replicated correctly across all clients is to re-implement the original algorithm that sent all keystroke messages to the server and storing them in a queue before sending them back to the clients. However, the high latency associated with this approach is less than ideal in musical performance where so many actions are time-sensitive. The alternative to this would be implementing a more complex algorithm for concurrent text editing, similar to the one implemented in Google Docs.

5.4 Phase 2: Operational Transformation

To combat scrambled text and high latency when communicating over the internet, the decision was made to implement a more sophisticated algorithm for managing multiple text inputs over a network. There is an algorithm called operational transformation (Ellis & Gibbs, 1989), which is implemented in many popular collaborative text editors such as Google Docs and SubEthaEdit, that ensures text consistency across multiple clients while maintaining high responsiveness of the system (Baumann, 2015). This section describes the implementation of the operational transformation algorithm and its effect on Troop in a performance setting.

5.4.1 Development

Operational transformation considers character insertions and deletions as operations on a matrix of characters. The server application maintains a copy of the shared document and keeps track of the changes, called operations, made by each client. An operation is a function that is performed on the entire text contents and returns a new version of the text with characters inserted or deleted. A client will apply any local changes immediately and then update the text using operations sent from other users in the order they were made. This allows the system to be responsive to a user’s interactions but also effectively preserve the order of text across all the connected clients. This is similar to the original algorithm I implemented but in operational transformation the server keeps track of all the operations and is able to combine local and incoming operations into a single operation, called a transformation, in the order they are processed and ensures that text is consistent for each connected client.

This was implemented in Troop using the Operational Transformation library for Python devel-

```

Here is some multi      000000000000000000000000
coloured text edited by 000111111111111111111111
three users in total.   2222222222222222

The numbers on the right 1111111111100000000000
correspond to the ID    2222222222222222222222
number of the user that 2222222222222000000000
added a particular      0000000000000000000000
character.              2222222222222222000000

The text that user sees 000000000000111111111111
is coloured depending on 111111111111111111
the ID of the user.

```

Figure 5.7: Representation of multicoloured text in Troop using user ID numbers.

oped by Tim Baumann, which is maintained at his GitHub². It contains modules for representing the client and server objects as well as one for creating and transforming text operations. This library handles all of the synchronisation aspects of the operational transformation algorithm as long as it is implemented in the program correctly. This complex task was further complicated by the fact that the colour of fonts was also required to be synchronised across the multiple clients. Previously Troop would simply insert a single character with a user’s associated font colour but operational transformation requires the text to be replaced with a new, transformed version and would consequently remove any colour information attached to it. Information about each character’s author needed to be explicitly stored and was achieved by using a second text document, hidden from the user, that applied the same incoming operations to it, but would insert the client’s ID number in place of each character. After this document was updated, the colours of the main text could be ‘refreshed’ using the ID numbers’ locations, illustrated in Figure 5.7.

The location of each user’s text cursor also needed to be calculated based on the incoming operation messages. Similar to text insertion in the previous phase of Troop’s development, a single character insert would move the cursor forward one place, and a single delete would move it backwards. As mentioned above, the operational transformation algorithm deletes the entire text contents when inserting or deleting characters, which causes all of the text cursors to be moved to the start of the document. To keep track of users’ text cursors, a function was written (see Figure 5.8) for calculating the location of the cursor after an operation was applied to the document. This function, along with another called `get_operation_size`, would make useful additions to Tim Baumann’s operational transformation library and help others working on similar projects. This code has been submitted to be added to the GitHub repository so that it will be made publicly available in the future.

Another change added to Troop was the ability to set the interface background to transparent,

²<https://github.com/Operational-Transformation/>, accessed: 20/02/19


```

def get_operation_index(ops):
    """ Returns the index that a cursor should be *after* an operation """

    # If the last operation is a "skip", offset the index or
    # else it just moves it to the end of the document

    if isinstance(ops[-1], int) and ops[-1] > 0:
        index = ops[-1] * -1
    else:
        index = 0

    for op in ops:

        if isinstance(op, int) and op > 0:
            index += op

        elif isinstance(op, str):
            index += len(op)

    return index

```

Figure 5.8: Python Code for calculating a user's location after a text operation.

which would allow it to overlay a visual accompaniment program. This was a result of a discussion with members of TYPE, which resulted in a desire for more ways to relate code to sound during performance. A menu option was added to interface that, when clicked, sets the background of each element of the interface to black and then removes all of the black pixels, thus making it appear transparent, as shown in Figure 5.9, with the aim of allowing Troop to overlay audio reactive software.

5.4.2 Practice

Algorave Assembly Lunchtime Concert, Leeds - 27/04/18

Video recording: `ch5_2-Algorave.Assembly-27_04_18.avi`

See Appendix A.2 for performance description.

Algorave Assembly was an event run by the University of Leeds as part of Leeds Digital Festival in April 2018. It consisted of day-time workshops and performances, talks, a panel discussion, and ended with a late night Algorave. It was funded by The Centre for Practice Research in the Arts (CePRA) and aimed to introduce A-level students to new forms of music making that they may encounter if they chose to study music at a university level.

As part of this event, TYPE was asked to play during both the lunchtime concert and the evening Algorave; and both performances were extremely different. I will be talking only about the lunchtime concert as I felt it was the more interesting of the two performances, mainly because it was not an Algorave performance itself and allowed us to express ourselves unconstrained by

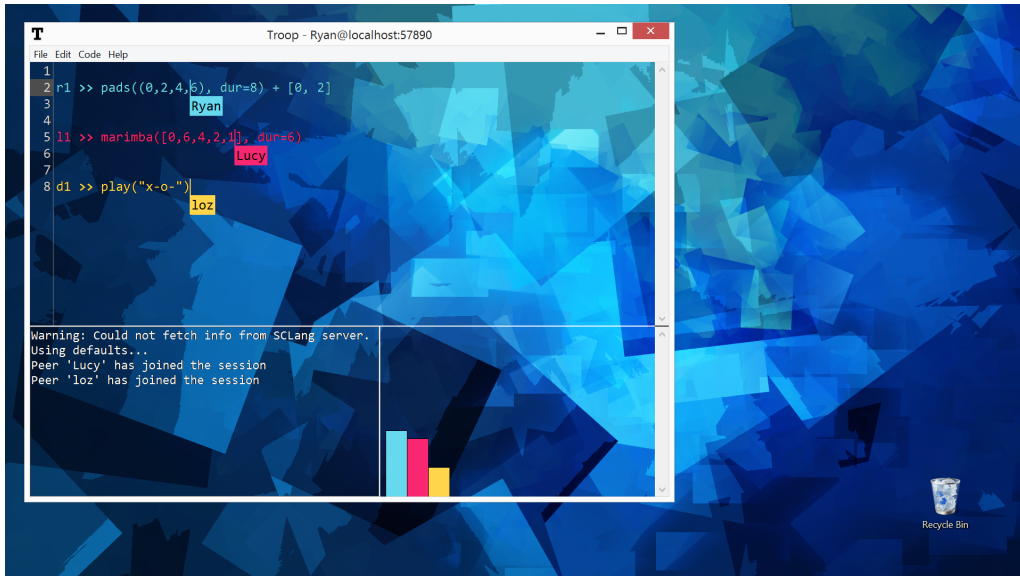


Figure 5.9: Troop interface with transparent background.

the expectation to make electronic dance music. As we were performing to an audience who were unlikely to have experienced live coding or Algorave before, we wanted to make the connection between code and sound as clear as possible. To do this we overlaid SuperCollider's oscilloscope, which displays the waveform for the audio being generated, with the Troop interface using the transparent background mode, as shown in Figure 5.11.

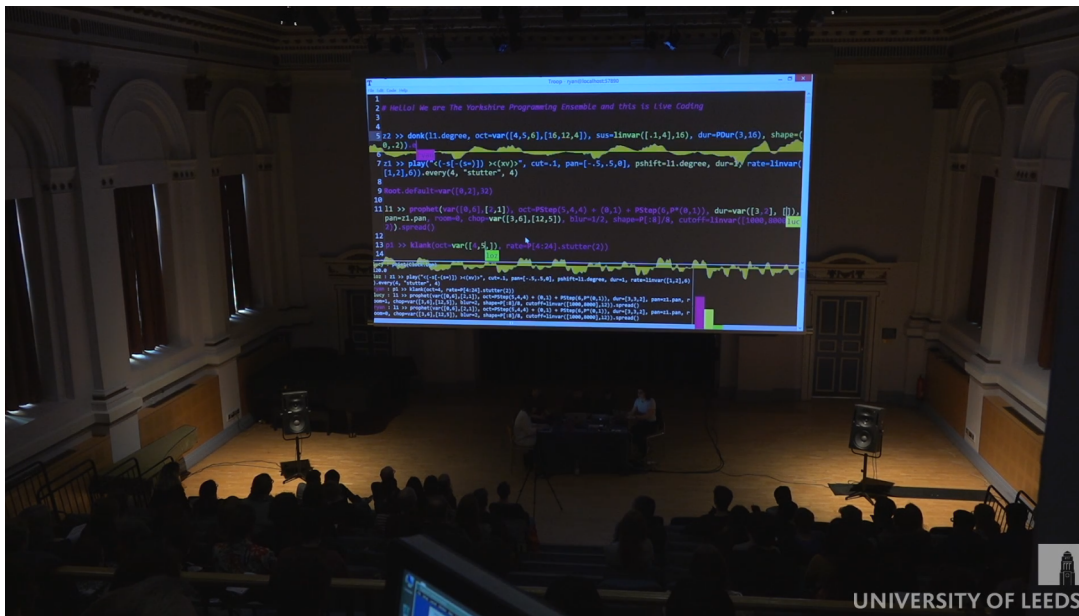


Figure 5.10: Still frame from the Algorave Assembly performance. Used with Permission from The University of Leeds.

5.4.3 Evaluation and outcomes

In the evening Algorave performance we aimed to make music for people to dance to by focusing on developing rhythms and repetitive musical patterns as an overarching aesthetic goal. Comparatively, we didn't have a goal in this performance; it was a free improvisation in the sense that there was no predefined material but we were, of course, constrained by the affordances of the FoxDot software. Having been playing together for over a year we were much more in tune with one another's performance habits and we were able to compliment each other's styles well while we navigated a much more open-ended performance. The introduction of the operational transformation algorithm was also beneficial to the performance as it made Troop more responsive and reliable, making it easier to produce a varied soundscape by fine-tuning the timbres and create more generative percussive rhythms.

Free improvisation performances often find themselves on the the atonal and arhythmic ends of the musical spectrum but our performance was still highly metric in its timing and featured several harmonic components. This likely reflected the number of Algorave performances we had done together in which experimental techno music had become our norm, as well as the musical limitations of the FoxDot software. Another contrast to more traditional free improvisation was the explicit discussion of emerging performance goals; we communicated the idea of moving into specifically defined sections, percussive and Algorave, by writing comments within the code itself. This formal agreement between performers to pursue an aesthetic goal midway through a performance is unconventional but the use of code comments shared the communicative process with the audience, who responded positively through laughter. As we entered the Algorave section we introduced more identifiable musical elements such as four-to-the-floor kick drums and a bass synth, moving the performance away from a free improvisation and closer to an EDM jam session. I think we were happy with this outcome as we hadn't set out to try and free improvise the whole performance and it felt right to show a little of the musical style we had been developing while performing on the Algorave circuit. We did not move completely away from free improvisation and indeterminacy, however, as we finished the performance by incorporating the sound of Super-Collider crashing into the closing of our set. In live coding you are often in complete control of the computer's actions but listening to the powerful sounds of the computer taking it back presents a striking dichotomy of the human and the computer in performance.

From a technical standpoint, one of the main issues with this performance was with Laurie erroneously being logged out then being assigned a new colour when logging back in again. This was caused by a network issue, which was likely a result of the machine hosting the Troop server application being overloaded. It was being run on the same laptop generating audio and the oscilloscope visuals, which probably starved the server application of computing power, causing it

to drop messages and log Laurie out. These sorts of issues are hard to plan for and will happen in live performances from time to time. After having to log into Troop a total of three times, Laurie's contributions to the code were separated into three different colours, one of which was the same as another user, as is the case in Figure 5.11. Coupled with the oscilloscope in the background, this created a very frenetic visual effect and probably did more to obscure what was actually happening in the performance. A useful feature to implement would be the ability to re-log into Troop and resume coding with the same coloured font.

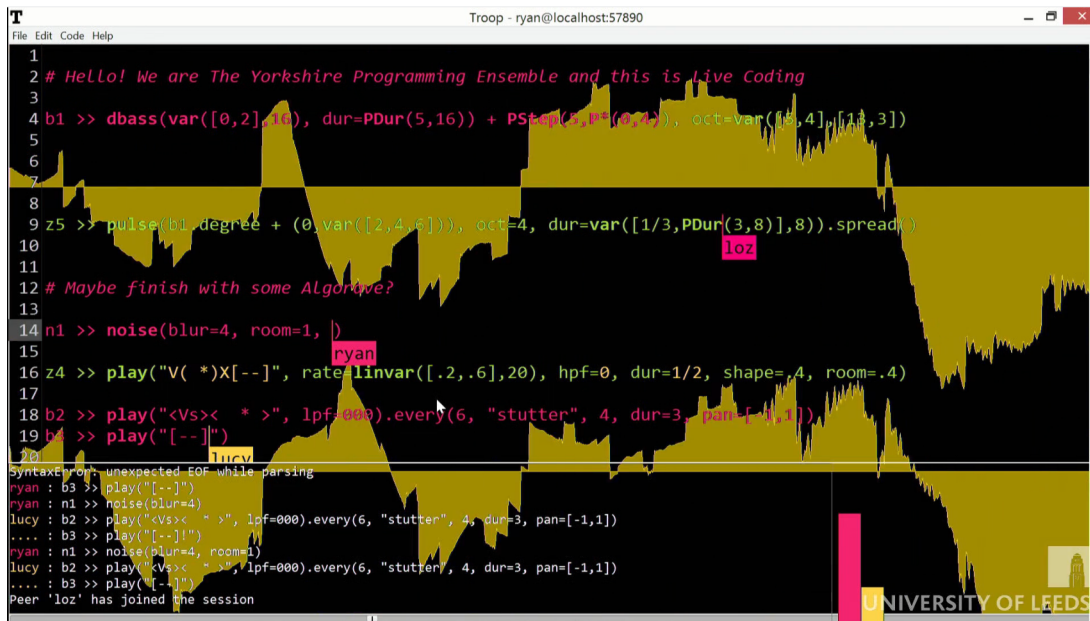


Figure 5.11: Troop interface overlaying on SuperCollider's oscilloscope.

The use of many different coloured fonts in this performance prompted discussion within TYPE about how the manipulation of these colours could be used in an aesthetic capacity. Over the course of a typical performance, the coloured text would become highly interwoven and this non-uniformity might give the impression that the group was lacking in synergy. It might reflect, inaccurately, that performers' creative thoughts were at conflict with one another, when the reality was that the more the coloured text was spliced together, the more we were playing off each other's ideas. This was definitely an idea we wanted to explore in performance and could potentially lead to aesthetic changes to future iterations of Troop regarding user's font colours.

Throughout its development, we have found that we have really enjoyed using Troop to combine our musical ideas in FoxDot but there are many different live coding languages available and it would be great to share this tool with as many live coders as possible. To do this Troop would need to become language agnostic in a similar vein to Extramuros. Currently Troop simply imports the Python code for FoxDot directly into its environment but this would not be possible for languages not based in Python, such as TidalCycles or Sonic-Pi. Adding the ability to interface with multiple different live coding environments would be a great way to introduce collaborative live coding to

more people.

5.5 Phase 3: Language Agnosticism

To bring collaborative live coding with Troop to as many live coders as possible, the decision was made to adapt the program and allow it to be used in conjunction with languages other than FoxDot. The section describes how this was implemented and the impact that this has had on the Troop project and the live coding community as a whole.

5.5.1 Development

In previous phases of development, Troop simply imported FoxDot into its own codebase using Python's `import` keyword. As this is not possible to do with other live coding environments, Python's `subprocess` module, which allows external programs to be run from inside a Python application, was to be used instead. A module was added to Troop called `interpreter.py` that defines the classes used for communicating with host live coding languages. Each one spawns a process and pipes text into it whenever code is evaluated in Troop, then reads any output from the process and prints it to the Troop console. Currently there are interpreter classes for the FoxDot and TidalCycles languages and a generic interpreter that could potentially be used with any language; if a program can take a raw string input and convert that to usable code then Troop will be able to communicate with it. Specifying the `mode` flag at the command line, followed by the name of the desired language, will start the client with a different interpreter to FoxDot like so:

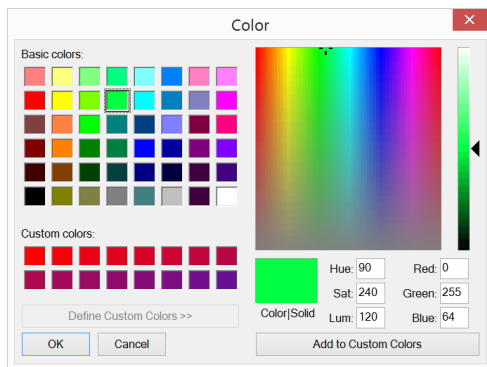
```
python run-client.py --mode TidalCycles
```

The interpreter can also be selected from a drop-down menu while Troop is running, which potentially means multiple languages could seamlessly be integrated into the same performance provided the set up is correct. By giving performers flexible access to multiple live coding languages Troop can enhance musical expressivity and broaden the channels for collaboration.

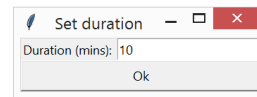
Not all live coding environments can take text as an input using Python's `subprocess` module. For example, this is the case for popular live coding environments SuperCollider and Sonic-Pi. These two particular environments, however, do allow for text to be executed when received over OSC. As well as interpreter classes created using Python's `subprocess` module, Troop's `interpreter.py` module contains classes for evaluating code over OSC. For SuperCollider and Sonic-Pi it contains language-specific information, such as the OSC address to send code to, but a general-purpose OSC interpreter also exists that would make adding more language-specific classes simple. Unfortunately, sending code over OSC is a one-way process and output from the external program does not get

printed to the Troop console.

As discussed in Section 5.4.3, there were some issues with users logging back into an existing session. If users disconnect from Troop they are treated as a new user when they log back and are assigned a new colour when they reconnect. To avoid this from happening, the Troop server now keeps an address book of all the users that connect and stores their login name and IP address. If a user is disconnected and then logs in with the same name and address, the server will re-assign that user their user ID and text colour. If two users connect with the same IP address (a feasible possibility if two users on the same local network connect to a server over the internet) and the same name then the second user to attempt to connect will be shown an error message.



(a) Colour palette for selecting merge colour



(b) Window for selecting duration

Figure 5.12: Windows used to start a colour merge sequence in Troop.

Also discussed in Section 5.4.3 was the idea that font colour could be explored as a way to highlight collaboration within the ensemble. During a typical performance with Troop, users will edit the code written by their co-performers, interweaving the performers' colours within the text. These combinations of colours are evidence of performer collaboration but, over the course of a performance, they can make the code look confusing and hard to read. The lack of uniformity in colour may also suggest that there is a lack of cohesive communication when, in reality, that is not the case. In an attempt to emphasise collaboration in Troop, a function to “merge” users' colours over the course of a performance was added. From the menu users can select a colour (Figure 5.12a) and set the duration required for the font colours to merge (Figure 5.12b), which activates the colour merge sequence. A computer represents a colour as three numbers between 0 and 255, with each number relating to the strength of the colour's red, green, and blue (RGB) channels. This means a colour can be represented as a point in a three-dimensional space whose co-ordinates would be equivalent to its RGB value. During the colour merge sequence, Troop moves each user's individual colour towards the colour selected by the user within this space. As this happens, Troop refreshes the colour of the text ten times per second with the updated RGB values, creating the appearance of the text colour naturally blending into one. The result can be seen in Figure 5.13, which shows three users editing code in Troop and their user colours merging into light-green over

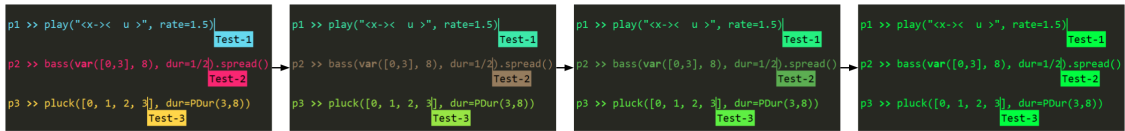


Figure 5.13: Progression of Troop's font colour merge.

time.

5.5.2 Practice

International Conference on Live Interfaces, Porto - 14/06/18

Video recording: [ch5_3-ICLI-16_01_19.mp4](#)

See Appendix A.3 for performance description.

Having been improvising together at late-night Algoraves together for over a year, TYPE wanted to explore live coding ensemble performance in a more structured and conceptual way. We had been playing with Troop's colour merge sequence tool and wanted to combine it with a framework for improvisation in order to accentuate the creative communication within the ensemble and demonstrate Troop's strength as a collaborative tool. We came up with an idea for a piece called *fingerprints* and submitted a performance proposal to the International Conference on Live Interfaces (ICLI), which was accepted shortly after. Below is part of the description taken from the proposal:

This is an improvised musical performance that explores the idea of ownership in collaboration. When working within a shared text editor the notion of ownership is continually brought into question as the textual material created by one performer is constantly reshaped by the other members of the performance. The performance begins with each live coder creating a separate strand of musical code before moving on to edit someone else's existing strand. They are then tasked with transforming what is considered code 'owned' by someone else into something that no longer resembles (sonically) the original algorithm. This act of reshaping a user's identity within the code creates a driving force of change and unpredictability. Music is still created with considerations towards the overall sonic output, so changes are gradual but definite. Each performer must let go of the idea of the 'self' and allow their work to be lost within the combined efforts of the group. As each musical strand is constantly in a state of flux, performers must react and adapt to the changes in real-time. The aim is that through these actions performers will no longer think about their contributions with a sense of the self but with a sense of communal action that allows the music to

emerge and evolve naturally from joint processes as opposed to being driven by a single user or musical impetus.

The piece *all voices are heard* (Saunders, 2015) by James Saunders explores group behaviours in a similar manner, drawing on research in the social sciences into “the similarity heuristic and consensus decision making”. It “asks players to compare performed sounds and make alterations until all players are playing the same material, and consensus is reached” where all the material is seemingly preexisting and performers choose the order in which it is played in repeated cycles until the order is eventually agreed upon. In fingerprints, however, performers are tasked with generating their own material and sharing it with one another over the course of the performance. fingerprints also draws inspiration from the piece, *Mind Your Own Business* (Hummels, 2013), written by Jonas Hummel for the Birmingham Laptop Ensemble, in which the ensemble starts with three synthesized sounds and each performer is given one or more attributes (rhythm, pitch, timbre, and manipulative effects) to explore as a group, which they alternate between over the course of the performance. Unlike in *Mind Your Own Business*, the performers in fingerprints are able to control any number of musical attributes but alternate between the strands of code in the shared editor itself. The Troop software itself has been repurposed for this performance to help emphasise the transition from individuality to group collaboration; by default each user is assigned a different coloured font to help identify the separate contributions but over the course of fingerprints each user’s assigned colour will converge to the same shade as to accentuate the coalescing of the performer’s minds.

The piece is an exploration into collaborative techniques for introducing high levels of variety in short spaces of time in collaborative live coding with Troop. It challenges the idea of ownership and authority and aims to create a completely democratised piece of music that still gives performer’s individuality. Once each performer writes a separate line of code, they “rotate” and work on one another’s code; changing it in a way that no longer reflects the original author’s idea but still fits within the overall aesthetic of the soundscape. You are constantly treading on each other’s toes but continually discovering novel musical ideas from the code created by others. The performance took place at the Passos Manuel bar in Porto and only I was physically present as Laurie and Lucy were connecting remotely via a publicly accessible Troop server. This demonstrated Troop’s potential as a powerful tool that enables international collaboration. Even connecting to a Troop server running in a different country, the interface remained responsive and accurate thanks to the operational transformation algorithm implemented in Phase 2.

5.5.3 Evaluation and outcomes

This ended up being a much more difficult performance than we expected. We approached it in a very different way than we had done before and this definitely placed us outside of our comfort zone. There was often a tension between following “the rules” and following one’s instincts while trying to perform without the “sense of self”. However, by focusing on a process driven by change, as opposed to an aesthetic goal, we explored a broad range of electronic textures and produced more challenging rhythmic patterns than we had done in previous performances. Continually re-inventing existing lines of code, as opposed to creating new ones, pushed the music into territories we wouldn’t normally go into in our typical performances. Interestingly, by approaching this performance with a specific process we probably incorporated more free improvisation techniques than we did in the Algorave Assembly performance in which we stated it was a free improvisation from the start. The focus on process helped prevent us from falling into old habits but it did feel restrictive and uncomfortable at times. While we agreed this performance was a worthwhile endeavour, we felt that we enjoyed more loose improvisation that encouraged self-expression and where ideas can be easily shared and developed using Troop.

As a conceptual piece, the process-centered style of performance with *fingerprints* worked well when combined with Troop’s colour merge sequence to highlight the collective coding as a single work of the ensemble as opposed to the sum of the individuals’ contributions. However, there are some aspects of the colour merge feature that could be improved upon. For instance, the final colour is not informed by any sort of performance data; it is arbitrarily chosen at the start of the performance and set to merge together over given period of time. It seems like a bit of a wasted opportunity to not have utilised at least the data available in Troop’s character-tracking graph to inform the colour merge in some way. The final colour could potentially have been calculated based on the average of all the user colours and weighted by their contributions to the code at each point in time. This would have still represented the collaborative nature of the work but also underlined the individual performances that come together to make it and would also have been a better semiotical tool for indicating to the audience what was actually happening during the performance.

During this phase the ability to use Troop with different live coding environments was added. There are many live coding languages, and many more popular than FoxDot and Troop should be a tool that allows any live coder to collaborate. After adding this feature it was great to see Troop being used by members of the live coding community. The live coding duo, Class Compliant Audio Interfaces (CCAI), use TidalCycles and drum-machines to perform and also utilised Troop for sharing code during a performance at Sheffield’s Access Space (see Figure 5.14). It is amazing to see Troop used in a performance, even before the project is completed, and it provides tangible

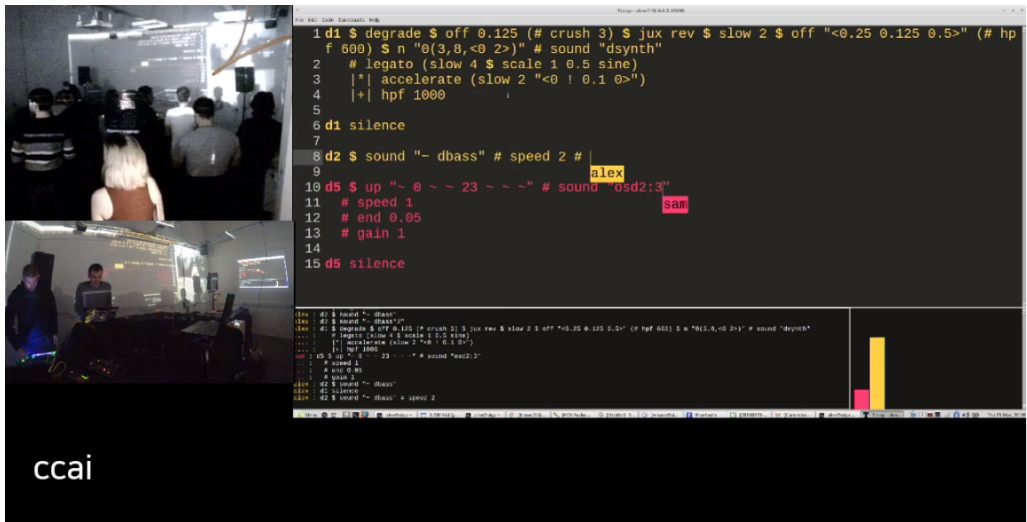


Figure 5.14: Class Compliant User Interfaces using Troop, Access Space, Sheffield, 15/03/18. From https://www.youtube.com/watch?v=3BWfPvdEy_o, accessed 20/03/18

evidence that Troop is a useful asset in ensemble performance.

5.6 Conclusions

5.6.1 Personal reflection

By separating users’ contributions by colour, the creative processes at play during performance are revealed to the audience as they overlap and intertwine. Augmenting the software to merge these colours over time emphasises the idea that the individual contributions of ensemble members are part of a greater whole. One of the research questions for this thesis is “how are collaborative interfaces used to reveal creative processes at play in ensemble live coding performance?” and I believe that Troop does reveal these processes very effectively. Being able to see, and directly work with, co-performers textual material opened exciting possibilities for ensemble interaction; during the performance at Leeds University in Section 5.4.2, it was very musically satisfying to combine code to create new ideas in the moment. This is something that would be very difficult to do without working with each others’ code directly, as you do in Troop. Similarly, being able to make comments in the code and communicate ideas, such as moving onto a new section, explicitly revealed creative processes to both members of the ensemble and the audience. The research question “how can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?” is explicitly addressed by allowing users to collaborate with one another directly in the code.

In terms of strategies for performing together with the tool, it would seem that overarching aesthetic goals were more successful than defining processes to be implemented in performance. Improvising freely with Troop, without any pre-assigned roles, often led to the discovery of novel

musical ideas that emerged from the combination of multiple performers' contributions to the code. Using a stricter framework that defines *how* one codes, such as the one implemented as part of the performance at ICLI in Section 5.5.2, potentially stifled creativity but were interesting to explore nonetheless.

An unintended benefit of using Troop as an interface for ensemble live coding was the ease of set up. Only generating audio from one laptop, which requires no audio synchronisation lets you approach set up with a “plug and play” attitude. There is very little configuration needed and only using one sound source also makes it a sound engineer's dream when performing as part of a larger line up with lots of acts; something very common in the Algorave scene. Combining this with the user friendly interface and easily distributed software, Troop facilitates ensemble live coding well, even before a performance has started. A secondary goal of this research is to lower the barrier to entry for collaborative live coding and reducing the time and effort required to set up for a performance goes a long way in achieving this.

There are also some negatives regarding the Troop interface. One issue that has been particularly difficult to address is that of syntax highlighting. Modern text editors, such as Atom³ or VSCode⁴, use coloured text to help make code easier to read, which would be useful when collaborating. However, because Troop uses different coloured fonts to differentiate users' contributions, it makes syntax highlighting an almost impossible task. Although the use of italics and bold are used to highlight comments and keywords respectively, it is not as effective as using colour to identify syntax. Furthermore, the use of the bar chart to represent total contribution by using each user's colours has also been an issue. In Section 5.3.1 using the amount of characters a user could type was discussed as a potential creative constraint for a future performance. The idea would be to stop one user from dominating the performance and give users with small contributions time to write code on their own and “catch up” to others. However, after using the Troop system a number of times TYPE developed good chemistry and a balance across all performers' contributions and felt that there was no need to implement a character-limiting constraint. It was a personal decision to leave the contributions graph as part of the interface as I was quite fond of its aesthetic quality and it still provided useful information about the performance to both the audience and the performers. It does mean that particular small (or large) contributions are made more explicitly clear and could, potentially, be quite embarrassing for a particular performer if they have a mental block and don't write code for a few minutes. That being said, this was never case during any of our performances, but it is something to keep in mind for, perhaps, ensembles made up of live coders with various levels of performing experience.

³<https://atom.io/>, accessed 25/10/2018

⁴<https://code.visualstudio.com/>, accessed 25/10/2018

5.6.2 User evaluation

As mentioned in Section 3.4, the interfaces developed over the course of this PhD will be evaluated through practice and by assessing key creative properties outlined by Gifford et al. (2017); the notion of “trust” in the interface, the ability to achieve “flow”, and the sense of “immediacy” associated with the interface. “Trust” here refers to “handing over of responsibility for some part of the music production to the computer” and having confidence that it will take some effective creative risks when doing so. The creative risks taken by the interface are actuated by the host language, such as FoxDot, as opposed to Troop itself so I will be assessing the level of trust in the interface to work correctly and allow users to take creative risks. At the end of this project I interviewed the members of TYPE and asked them to discuss the concepts of trust, flow, and immediacy with regards to using Troop and it raised some interesting points. It should be noted that at this point TYPE now had the addition of Innocent Granger, whose contributions are more notable in the following chapters, as well its original members of Lucy Cheesman, Laurie Johnson, and myself. When asked about how much trust they had in Troop to allow them to take creative risks in our improvisations, Lucy gave a very insightful response:

I guess the trust level is more, like, about how we play together because I feel like if I dropped something, like, that’s a bit of a clanger, it’s not gonna, like, stress you guys out. [...] I feel like something that’s really nice about Troop is that I can kinda see what everyone else is working on really easily, so, if I see that, like, ‘oh, actually Ryan’s writing a whole new player’, then I better, like, not change something major ‘cause it will, like, clash with what he’s doing. Or if I see that people are just, like, tinkering round with stuff, then I’m, like, ‘oh I can do a major change here’. So it’s, like, really immediate in that sense of, like, I can see exactly what everyone is up to and, erm, then I sort of can then – in terms of, like, taking risks – will try stuff out I know when it’s gonna be an appropriate time to do that.

When asked if that means she feels that trust in her co-performers was more important than trust in the software, she replied “Yeah I think so but I think Troop allows that because it allows me to kinda see what you guys are up to and also allowed us to practice together loads in a really, like, low-maintenance and low-effort way, which has enabled us to build that trust up”. Trust between ensemble members is very important in improvisation and this holds true for live coding. Interestingly, Troop seems to strengthen the level of trust between performers by giving them a real-time overview of the process of coding and helps inform performers’ creative decisions, such as when musical changes should be introduced. Practising together over the internet also enabled us to build up trust and we were able to do this much more frequently than if we had had to rehearse in person. Lucy referred to these online practices as “low-maintenance and low-effort”, which was

one of the goals of this entire PhD; making ensemble live coding as easy and accessible as possible.

Innocent said “I trust the software to do what I expect it to do” but he also stressed the importance of trusting himself “to know that [...] I’m not gonna play something that’s just gonna make everyone’s ears bleed”. Along with trusting co-performers, there is also a strong feeling of responsibility and self-awareness among some members. He goes on to say that “every time Troop’s fallen over in our performances it’s more down to, err, human error than the software itself. So, umm, yeah either playing too many, uhh, things at once, or just pressing stop”. The latter example he provided is a reference to the occasions where one of the ensemble members accidentally triggers the keyboard shortcut for clearing the scheduling clock. Laurie feels like these instances of human error are easier to overcome with “four pairs of hands” but goes on to say that the error itself is “like a refresher as well, it just strips- the music takes a break for a bit and layers die away kinda gradually, not- it doesn’t just stop instantly but actually works quite well”. This is particularly interesting as it demonstrates that there is trust in the interface, in the sense that Gifford et al. discuss it, that it will support us in a creative manner in the case of failure. I then asked the group if they managed to achieve a sense of flow when performing with Troop and, again, Lucy provided some useful insights:

I would say, like, when compared to Tidal, like when I’m playing gigs on my own, if something goes wrong in- when I’m playing with TYPE, it... it takes me out of it less because... I think because it’s easier to recover because there’s more of us. It’s like not all on me it’s like if I, if I, you know I, like, if something goes horribly wrong, you guys can help fix it and recover and, like, turn in round, in a way. And I don’t if that’s – I mean obviously it’s partly to do with Troop because obviously we’re all working together and we can all, like support each other in, kind of like, correcting any errors – but I also think, like, from a kinda confidence and just, like, capacity perspective, erm, I think it’s easier to, sort of, stay in the flow and to keep things going with Troop compared to playing solo, definitely. And I think that’s a combination of the software and also just of the nature of playing with other musicians.

This sentiment was shared by the rest of the group. Again it seems that the act of playing as an ensemble with a strong sense of trust played more of an impact than the interface. Performers did achieve a sense of flow when performing with Troop but this was mainly due to the confidence in their co-performers, on whom they could rely to help deal with problems such as errors. It is difficult to say, at this point, whether Troop is facilitating this sense of flow without a direct comparison with other collaborative interfaces, but future chapters will enable this with potentially fruitful results.

Lastly, I asked whether they felt they had a sense of immediacy when using Troop regarding

both their own musical expression and their ability to communicate ideas with the rest of the ensemble. For Innocent, confidence in your own ability was key to musical expression. He said “if there’s something happening and you think of something you can immediately get it out there, umm, it just depends, I guess, on your level of confidence in terms of how... how confident you are to play that without any sort of guarantee of how it’s gonna sound”. As with any method of music-making, it is difficult to perfectly recreate a musical idea in your head with the instrument in your hands and, without confidence in your own ability, you may feel like your new musical idea may be more disruptive than constructive in an ensemble context. Because of this, many changes to code, and consequently sound, are incremental during performance, perhaps not giving users a true sense of immediacy. Lucy confirms this by saying “I also think that, like, in one sense, it’s very immediate but in another sense I’m more likely to, like, start with a simple idea and build it up”. When asked about the ability to react to their co-performers’ musical changes, Lucy stated:

I find Troop pretty good for stuff like that because you can see what everybody’s up to. So, like, I can see if, like, someone is creating a new player at the same time as me and I can kinda feed off that to an extent, erm, in a way that can be a bit disruptive sometimes but it’s quite easy for that disruption to be positive because I can, you know, I can take elements from somebody’s pattern or something like that. So I would say personally I find that quite, erm, a positive feature in Troop.

The incremental nature of self-expression in Troop allows users to see, as well as hear, the changes being made to the code over longer periods of time. This allows the performers to adapt to any “disruptive” changes in the music in a way that takes the improvisation to interesting new places. The collaborative act of coding seems to invariably lead to disruption in musical performance, but our coding styles have adapted to this by only making small incremental changes to the music where possible. This, coupled with the ability to see each others code as it is being written, consequently helps inform our own musical decisions. With Troop we actively try and avoid disruption but it invariably occurs regardless and this seemingly enables fruitful musical ideas to develop as a response.

5.6.3 Quantitative evaluation

Over the course of the project Troop has seen an uptake in use within the live coding community and this larger sample size of users has allowed me to collect user feedback through an online survey. The data was collected using a Google Form that was available from Troop’s GitHub repository page. All responses were anonymous. The survey was adopted from a user satisfaction questionnaire proposed by Chin et al. (1988) and has been used to evaluate user satisfaction of the live coding language, *ixi-lang* (Magnusson, 2011b). A total of 7 responses were collected between

19 March 2018 and 19 February 2019. The repository has been ‘starred’ by 90 GitHub users but only ‘cloned’ by 12. Given that the number of users who have used Troop extensively is likely between those numbers, 7 responses is an estimated 15-20% response rate. The survey contained 37 questions, many of them requiring qualitative answers, but a small subset of these asked users to rate various facets of the interface on scale between 1 and 9. Results can be seen in Figure 5.15. All respondents found Troop to be wonderful as opposed to terrible and easy to use as opposed to difficult, but some felt that it was inadequately powered and inflexible. This was not the case in every instance as at least one respondent rated each attribute 9 out of 9.

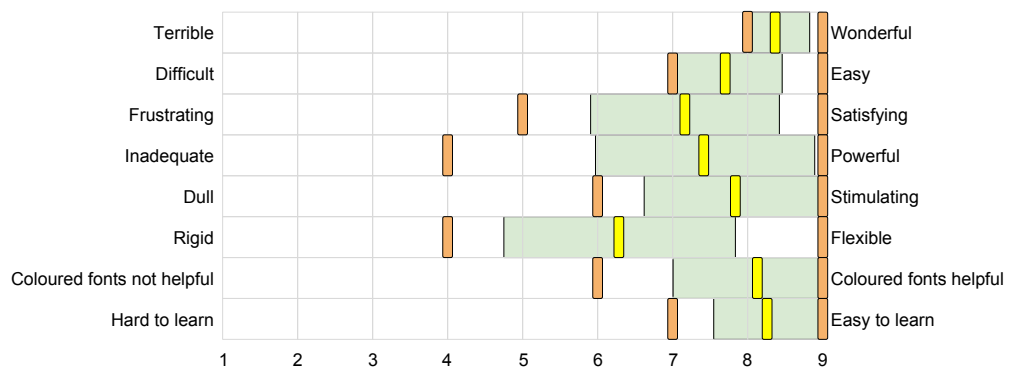


Figure 5.15: User satisfaction response graph. The yellow line is the mean, the green block represents the standard deviation, and the orange lines are the lowest and highest responses.

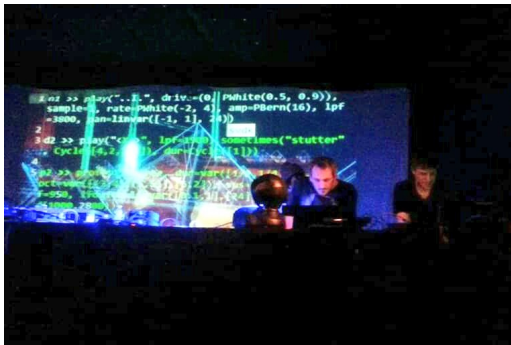
Users were also asked if using Troop to collaborate changed the way they live coded. Two users felt that it had no effect on how their style of live coding but five out of the seven respondents did feel that the interface affected the way they approached their practice. One user (A1) wrote “It gives me time to reflect upon changes in the sound because with other users making changes there is no rush for me to evolve the sounds”. This is not necessarily a direct result of using Troop as an interface for collaboration but from the act of collaborating itself. Live coding is a cognitively demanding tasks and requires listening, thinking, planning, and typing skills, and it is difficult to manage it all simultaneously in front of a live audience but spreading the load across an ensemble allows performers to allocate more time to each facet of performance including, arguably the most important, listening.

Live coders are not only having to apply multiple skill-sets during performance, but they also have to manage different musical aspects too, such as rhythm, melody, sound design, and form. Live coding as an ensemble allows performers to work on specific aspects of the music that they would not otherwise be able to dedicate as much time and effort to. This was clear in one user’s (A5) response, which was “In a team scenario, one is able to focus on one’s strengths when solo performance requires you to cover all aspects of live coding performance I.e creating new tracks, editing/enhancing tracks, transitions, etc”, suggesting that playing together as an ensemble allows the group to maximise their potential by having the individuals focus on the musical features they

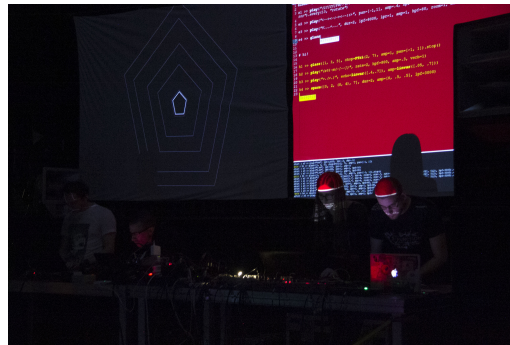
are most comfortable with.

One user (A4) did specifically mention how using the Troop interface with other live coders influenced how they would perform; “I think about things differently when collaborating with others – with troop you can see how other people are developing the code which in turn influences the decisions I make”. Being able to see co-performers’ code being written in real time gives users an indication of what is about to happen next and allows them to adapt their own code in response; realising one of the main goals of the Troop project.

The responses above demonstrate that live coding as part of an ensemble seems to reduce the cognitive load for individual performers, giving them more time for reflection and allows them to focus on their individual strengths. Furthermore, utilising Troop within an ensemble also helps give live coders insight into their co-performer’s actions and helps facilitate collaboration and communication in live coding ensemble performance.



(a) Crash Server.
Photo courtesy of Crash Server.



(b) Kolmogorov Toolbox.
Photo by Tatiana Soshenina.

Figure 5.16: Photos of Troop being used by various ensembles.

5.6.4 Impact

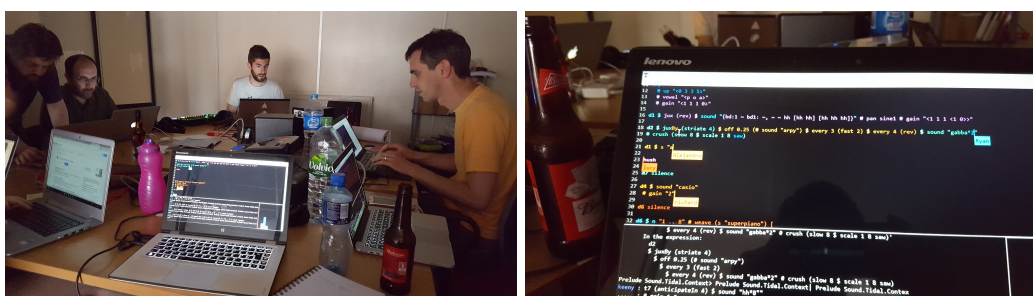
Making collaborative live coding simple and giving those with limited technical knowledge the opportunity to live code music as a group has been a key consideration while conducting this research. However, Troop has also had an impact among the more established members of the live coding community; it was shown in Section 5.5.3 that Troop was utilised by the Algrave duo, Class Compliant Audio Interfaces, of which one half is Alex McLean who is credited as co-creating the Algrave movement. Troop was also used by Hungary-based duo, Kolmogorov Toolbox, in a performance at the International Conference on Live Coding 2019⁵, arguably the largest live coding event in the world, and demonstrates that Troop can be used as a tool for collaboration at the highest level of live coding performance. Troop’s overseas impact has also been demonstrated by the French live coding duo, Crash Server, who use it regularly to perform with. Photos from

⁵Conflicting Phases - Kolmogorov Toolbox. <http://iclc.livecodenetwork.org/2019/programa.html#pn19>, accessed: 25/01/2019

performances of both these acts can be seen in Figure 5.16.

5.6.5 Potential in pedagogy

Music has been used as a useful teaching analogy for computer science in younger age groups, as demonstrated by software such as Scratch (Ruthmann, Heines, Greher, Laidler, & Saulters II, 2010) and the live coding language Sonic-Pi (Aaron, 2016), and collaborative live coding can help move student thinking “from an individual to a social plane” (Xambó et al., 2016). Troop was used as part of a live coding workshop in Sheffield in March 2018 in a purely pedagogical, as opposed to a performative, context. When adding the ability to use different languages with Troop during Phase 3 of development, I had very little knowledge of any environment other than FoxDot and I wanted to test how well the interface functioned when being used with an alternative language. A TidalCycles workshop takes place every month in Sheffield, called Tidal Club⁶, and organisers were happy to use Troop as part of the session. As someone who has very little knowledge of TidalCycles language, I assumed the role of student as opposed to teacher and found that Troop was a useful tool for group learning. More experienced users could implement more complex compositions of functions and fix syntax errors to allow for a smoother coding process for less experienced users like myself. Another useful benefit of using Troop as a teaching tool was that the workshop leader could write code to demonstrate how to use a specific function and then users could immediately change values and listen to how it affects the sound. In a typical workshop session, users would have to read the text off a screen or projector and write the code themselves, which would be susceptible to syntax error if even one character was copied incorrectly. A teacher would then have to visit each student that had errors and go through the problem several times. Using Troop, however, allows for teachers and workshop leaders to address these issues quickly and directly using the program.



(a) Using Troop with TidalCycles. (b) Close up of with multiple users connected.

Figure 5.17: Troop used at Tidal Club, Sheffield.

Collaborative software development through the use of “paired programming” has become popular in the industry as a style of agile development and has also been explored in a remote

⁶<https://tidalclub.github.io/>, accessed: 28/02/19

context (Flor, 2006). Education has followed suit and employed paired programming in teaching software engineering and studies have found that working closely with others meant that students were “working harder and smarter on programs because they do not want to let their partner down” (Williams, 2001). Specific tools have even been developed for managing larger-scale software projects, such as GitHub Classroom⁷, with a view to making computer programming education a collaborative practice. Troop has the potential to be utilised as teaching tool in schools for both paired programming and larger groups of students too. Live coding provides an exciting and engaging topic of study for teaching computer science and moving it into a collaborative context could provide a better and more enjoyable learning experience, as has been the case in the paired programming studies and using real-time collaboration software, such as Google docs, in an pedagogical capacity can provide “a more productive learning system” (Google Inc., 2018).

5.6.6 Final thoughts

This chapter has demonstrated Troop’s strengths as a tool for facilitating collaboration in ensemble live coding. It is also clear that TYPE have started to develop a way of playing together, beginning to generate tacit knowledge of the system and how they tend to interact with one another. Troop has been adopted by several ensembles across Europe and received positive feedback in a user satisfaction survey. Working with the same textual material allows live coders to easily share and combine ideas to create novel musical sequences. But how does the live coding language itself affect how these musical ideas are communicated during performance? Is it possible to develop a language that enables better and more responsive sharing of musical information when coding within in the same body of text? The following chapter explores these possibilities in more detail in the hopes of further improving ensemble performance in live coding with Troop.

⁷<https://classroom.github.com/>, accessed: 01/05/2019

6. Developing a Language for Live Coding in Ensemble Performance

6.1 Introduction

This chapter introduces several features that were added to the FoxDot live coding language during the development of the Troop interface, discussed in Chapter 5, to improve musical collaboration during improvised ensemble performance. Troop facilitates real-time cooperative work by giving users access to a shared text editor, but it seems necessary to address how properties of the code itself can afford different levels of collaboration. For example, collaboration can occur at an inter-personal level when performers write human-readable comments in the text or when editing a partner’s existing code – but is any collaboration occurring at an inter-musical level? This chapter posits the question, “how can a programming language be developed to facilitate inter-musical communication in live coding?”. This question will be addressed first from a computer science perspective and potential computer programming paradigms will be discussed. It should be noted that it is beyond the scope of this project to compare every programming paradigm that exists so only the two most prevalent in live coding environments, functional and object-oriented, will be discussed.

Functional programming is utilised in popular live coding languages such as Tidal (McLean & Wiggins, 2010), which is embedded in Haskell, and Extempore (Sorensen, 2011), which is based on the Lisp-based language, Scheme. As the name suggests, this style of programming focuses on evaluations and transformations of mathematical functions, which hold no state. In contrast, object-oriented programming, used in SuperCollider (McCartney, 2002) and FoxDot, is often wholly dependant on data-types, known as objects or classes, that hold a state that can be changed or accessed by functions that are specific to that class, called methods. While the use of mathematical functions to compose music in real time is a unique and exciting practice, object-oriented programming is often used to represent complex and real-world systems (Kindler & Krivy, 2011) and it could be argued that modelling inter-performer interaction through text would be more appropriately addressed through an object-oriented programming paradigm.

FoxDot is based in the Python programming language, which is primarily an object-oriented language but also supports other programming paradigms, such as functional and procedural. Due to the focus on object-oriented programming in FoxDot, it would make an appropriate host language for modelling inter-personal musical relationships. The Troop interface also primarily

uses FoxDot as its interpreter and has been the language of choice for performances by TYPE since its creation. The goal of this chapter is extend the functionality of the FoxDot environment to allow multiple performers to efficiently and effectively share musical information when working together using Troop. This should be accomplished through the use of interdependent relationships created using code that tie together the separate strands of music produced in improvised ensemble performance.

6.2 Phase 1: Modelling Interpersonal Musical Relationships

6.2.1 Development

FoxDot began development in 2015 with the goal of creating “dynamic musical systems” in solo live coding performance (see Section 4.2) and the library will be expanded here in an attempt to model inter-personal musical relationships that can be implemented using the Troop software or otherwise. FoxDot uses “player objects” that are given instructions by the user for different musical attributes, such as pitch or duration, that are continually iterated over until stopped. Player objects store the data for these instructions and the last musical event played can be accessed at any time by a user or another player object. This model allows for inter-dependant relationships between player objects to exist where musical attributes for one player object can be derived from another by accessing its state. For example, the pitch of one player, `p1`, is determined by the pitch of another, `p2`, by instructing `p2` to check the state of `p1` for its pitch value when triggering a sound in SuperCollider. This data is stored in player object’s attribute dictionary, called `attr`, which holds keys, the shorthand names of the musical attributes, and their corresponding values, sequences of numbers representing musical information. To access a specific sequence from the dictionary, such as the pitch, a user only needs to type the name of the dictionary followed by the name of the key as a string in square brackets. For example, accessing the pitch data from a player object called `p1` would be achieved by typing `p1.attr[‘pitch’]`. This can be utilised during a performance to create references between player objects in the code such that one player is seemingly playing along with another. However, this is only possible if both players are also using the same duration as well as pitch, which could be shared by referencing `p1.attr[‘dur’]`. This is a very verbose way of connecting two player objects and is also restrictive and unrealistic. The goal was to model real-world inter-musical relationships, which may share some aspects but vary others, and this does not. Another problem with this method was that the relationships were not *reactive*; if the pitch of `p1` was changed then `p2` would still be referencing the old values and the relationship would have to be updated by re-evaluating the code referencing `p1.attr[‘pitch’]`. Managing this during a live performance would be time-consuming and would add to the already taxing cognitive load

that live coding puts on a performer.

The process of modelling more realistic inter-musical relationships began by adding functionality to the player objects that would better recreate basic behaviours of improvising musicians. This began by implementing a method called `follow`, which allows one player object to follow the pitch of another while using a different duration. When a player object calculates the values with which to trigger a sound, it checks if it has been assigned another player object to follow and then accesses the current state of that player object to retrieve the pitch value. The code in Figure 6.1 exemplifies how the `follow` method can be used to define a relationship and be combined with simple addition to create harmonies.

```
p1 >> bass([0, 1, 2, 3], dur=4)
p2 >> pluck(dur=1/2).follow(p1) + [2, 4]
```

Figure 6.1: FoxDot code using the `follow` method in which a player, `p2`, is following another, `p1`, and alternatively increasing the pitch by two and four steps to play notes a third and fifth above the note played by `p1`.

The pitch from `p1` is being accessed every time `p2` sends a message to SuperCollider, so `p2` is always using the correct pitch value even if the data in `p1` is changed. This is a huge improvement on referencing the data directly via the attribute dictionary as now the relationship is much more reactive and works independent of the rhythm of the player objects.

6.2.2 Practice

Rehearsal sessions, various locations - 26/04/17 & 06/06/17

Video recordings: `ch6_1a-Rehearsal-26_04_17.mpg` & `ch6_1b-Rehearsal-06_06_17.mpg`.

See Appendices A.4 and A.5 for performance descriptions.

This phase of development coincided with the early development of the Troop interface, which was being tested regularly through practice sessions with members of TYPE. Our first performance together took place on April 28, 2017 but we did not make use the `follow` method during it. As noted in Chapter 5, most of the initial sessions using Troop were run as workshops and I would introduce the ensemble to different facets of FoxDot. One of the most recent additions to the language was the ability model musical relationships using the `follow` method but it was only introduced a number of days before the first performance and we did not utilise it. We did not perform together again for a number of months after this as both Troop and FoxDot were being further developed and tested. Due to this, I am only able to include extracts from rehearsal sessions during this period.

```

1 Scale.default = "minorPentatonic"
2
3
4 z1 >> bell([0,0,[0,(4,7)],-4], dur=PDur())
5
6
7 s1 >> play(P["S[xS]x-"].layer("mirror"), dur=PDur(5,8), pan=(-1,1), sample=(0,1), chop=[0,0,320])
8
9
10 print SynthDefs
11
12 l1 >> dirt().follow(z1) + var([0,2,4],2)

```

```

Ryan : s1 >> play(P["S[xS]x-"].layer("mirror"), dur=PDur(5,8), pan=(-1,1), sample=(0,1))
Ryan : s1 >> play(P["S[xS]x-"].layer("mirror"), dur=PDur(5,8), pan=(-1,1), sample=(0,1), chop=32)
loz : z1 >> bell([0,0,[0,(4,7)],-4])
lucy : l1 >> dirt().follow(z1) + (0,2,4)
Ryan : s1 >> play(P["S[xS]x-"].layer("mirror"), dur=PDur(5,8), pan=(-1,1), sample=(0,1), chop=[0,0,320])
lucy : l1 >> dirt().follow(z1) + var([0,2,4],2)

```

Figure 6.2: Screenshot from a rehearsal session.

6.2.3 Evaluation and outcomes

One of the common themes that arose during rehearsal was that the `follow` method was used almost exclusively at the start of the session, or at the beginning of a new musical section. This was not something explicitly discussed by the group and is an example of tacit knowledge of FoxDot; not just knowing the syntax but the intangible development of *how* it has been used in performance. It allowed performers to work on separate strands of musical code without having to worrying about clashing melodies or accompaniments. If one user starts writing a line of code that involves pitch data, another can just implement the `follow` method and add a list of values with the relative certainty that the two sequences will compliment each other. This process also means that users do not have to worry about continuously updating their code to accompany the changes made elsewhere as the pitch data of the player object using `follow` will always use the most up-to-date data. It has proven a useful tool for collaborative live coding in Troop but there are still issues regarding its flexibility in practice.

```

p1 >> pads([(0, 2, 4), (3, 5, 7), (4, 6, 8)], dur=[8, 4, 4])
p2 >> bass(dur=1/2).follow(p1)

```

Figure 6.3: FoxDot code using the `follow` method with chords.

The `follow` method was developed so that musical layers created by different users could be connected quickly and easily using Troop and this is achieved for the most part but only allows for musical sequences to be associated in the pitch domain. Other attributes, such as rhythm or amplitude, can not be followed unless the values are copied and pasted from one line to another or referenced using a player's attribute dictionary. As mentioned previously, these methods are more

time consuming to manage and do not create a dynamic relationship that is reactive to changes made to code.

The `follow` method also fails to create a relationship that is flexible to multiple notes being played simultaneously. Examining the code in Figure 6.3 for example; a player object, `p1`, is playing a very simple chord sequence based on triads and another player object, `p2`, is connected to it using the `follow` method. The bass player object, `p2`, cannot play the root note of the chords in isolation as the `follow` method forces all of the pitch values to be shared between the player objects such that there is no way of accessing a specific subset of these values. To create the relationship where `p2` only plays a single note, the sequences would have to be rewritten such that `p2` contained the pitch information and `p1` was told to follow it, adding a triad of notes to create the chords. Doing this in the middle of performance would be time-consuming and would require rewriting other users' code; defeating the entire point of using language as a means to facilitate collaboration.

To enhance FoxDot's ability to create inter-musical connections within code, the player object relationships need to become much more flexible; a connection should be able to be made between any attribute, not just pitch, and attributes should be able to undergo transformations, such as multiplication, subset creation, or conditional mapping of values.

6.3 Phase 2: Player-Key Data Structures

The best approach for meeting the requirements outlined in the evaluation of Phase 1 is shifting the behaviour away from the player object but to a new class entirely. Instead of creating relationships between player objects, users should be able to create relationships between player attributes, which can then be used by the player objects. This approach also enables relationships to exist between attributes of the same player object, which can be used to create some interesting self-referential musical ideas. This object is called a "player-key".

6.3.1 Development

A player-key is a data structure that holds a value, or group of values, that can be updated externally. When a mathematical operation, such as multiplication, is applied to it a new "child" player-key object is returned that stores a reference to its "parent" and the operation used. When accessed, the child player-key checks the value held by its parent, applies the operation, and returns an up-to-date value. For example, in the flow chart shown in figure 6.5, a new variable called `myValue` is created by accessing the pitch value for the player `p1` and adding 4. This creates a child player-key, which keeps a reference to its parent, `p1.pitch`, and stores an adding function and the value 4 so that when its value is needed it simply calls the adding function on the current

value of its parent and adds the value 4.

```
p1 >> pluck([0,4,[7,10],9], dur=p2.dur)
p2 >> sitar(p1.pitch + 4, dur=[1,1/2,1/2])
```

Figure 6.4: FoxDot code using the player-key data structure.

The player-key’s behaviour could be thought of as a form of reactive programming or lazy evaluation that allows for dynamic changes in a programs state. For each attribute in a player, such as pitch, duration, or amplitude, there is a respective player-key data type that is being updated internally. It should be noted that the `follow` method still exists, but simply gives a player object a reference to the ‘followed’ player object’s pitch attribute player-key. This retains the behaviour exhibited by the `follow` method created in Phase 1 but now other attributes can be ‘followed’ by referencing them when giving a player object instructions. The example in Figure 6.4 shows code that define two player objects, `p1` and `p2`, that follow the other’s pitch and duration attributes.

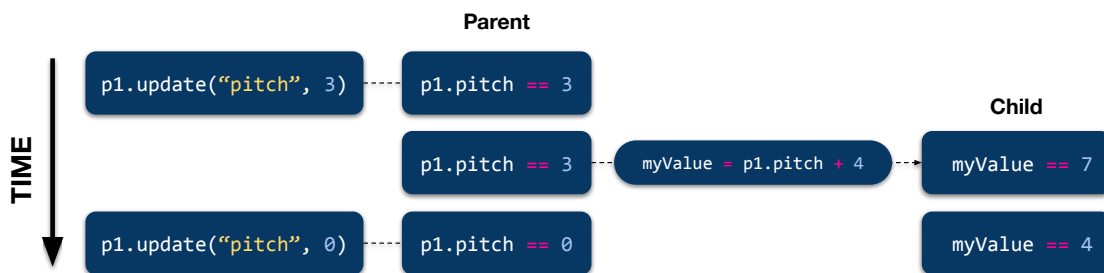


Figure 6.5: Flow chart diagram of psuedo-reactive player-key data type.

As mentioned above, a child player-key holds a relationships to its parent when a mathematical operation is applied to them. This is done by overriding the methods associated with mathematical operators in the player-key class to return a child player-key. Figure 6.7a is an example of operator overloading; the `__add__` method (called when the ‘+’ symbol is used with a class) returns a new player-key that contains a reference to the original (the `self` variable) and a function for adding the two values together. Performing a second operation on a child player-key will create another child player-key that, when accessed, will look to its parent, which will in turn look to its own parent, to return the value. A player-key can have any number of children and any child can have its own children. This creates a tree structure wherein each player-key is a node and any child player-key links back to the root parent, as demonstrated in Figure 6.6.

One of the shortcomings of the `follow` method was the inability for player objects to access individual items from a group of values, such as a chord, when following other player objects.

Just as the player-key's methods for addition and subtraction can be overridden to return a child player-key, so too can the item access method, known as the `getitem` method (see Figure 6.7b). A chord is stored as an array of pitches, e.g. `chord = [1, 3, 5]`, and any item in the array can be accessed by using square brackets and the correct index. The correct syntax to retrieve the first item in the array in this example would be `chord[0]`, which would return the value of 1. By treating the `getitem` action as a function, a child player-key can be created that will always hold a single value from a player object's attribute. Instead of just returning the first item in the array, it returns a player-key that checks the array's contents, which is updated whenever the player object plays a different chord, and returns the first item of the new array. Figure 6.8 is a snippet of FoxDot code for creating a bass that plays the first note from a chord sequence played by another player object.

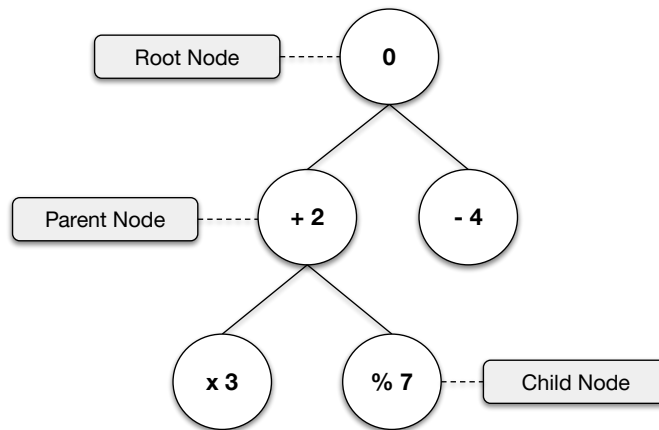


Figure 6.6: Tree structure relationships of parent and child player-key data structures. Each child node also stores the function that was applied to it.

In addition to the use of mathematical operators and the `getitem` method, player-keys can also be used in conjunction with basic logic. That is, when a player-key is asked if it is equal to, less than, or greater than another value, it will create a child player-key whose value will be 1 when the value of the parent player-key satisfies the statement and 0 when it does not. The code in Figure 6.9 shows how the “equals” operator can be used to create conditional behaviour; the second player, `p2`, will only have an amplitude of 1 when the pitch of `p1` is 4 and will be silent otherwise.

These logical tests can be combined with other mathematical operators to create dynamic conditions. For example, a user may want to pan a player to the left stereo channel by setting the pan attribute to -1 whenever the pitch of another player is equal or greater than 4. This can be done by using the ‘greater than or equal to’ operator, which uses the `>=` symbol, to create a new child player-key and then multiply it by -1 to create a child of that player-key, as shown in Figure 6.10. This idea can be taken further by also panning the player object to the right channel

```
def __add__(self, other):
    parent = self
    function = lambda value: value + other
    return PlayerKey(parent, function)
```

(a)

```
def __getitem__(self, index):
    parent = self
    def function(value):
        if isinstance(value, PGroup):
            return value[index]
        else:
            return value
    return PlayerKey(parent, function)
```

(b)

Figure 6.7: Simplified Python code for overloading the (a) addition operator and (b) element access method for the player-key data type.

when the pitch value is less than 2. To do this, another logical condition is applied to the same player-key and then added, as shown in Figure 6.11.

```
p1 >> pluck([(0,2,4), (2,4,6), (3,5,7), (4,6,8)], dur=4)
p2 >> bass(p1.pitch[0], dur=1)
```

Figure 6.8: FoxDot code using player-keys to select single notes from a chord.

The combination of conditionals and multiplications can be considered a mapping function from one player-key to another but its complexity increases every time a new possible output is added. To simplify mapping player-key values to specific output values, a method called `map` was created that allows a user to specify specific output values for a given set of input values. The method takes a dictionary of values and creates a new function that is supplied to to the child player-key, as shown in Figure 6.12. For each value in the mapping dictionary the function will test if the parent player-key is equal to it, and return the corresponding output value if it is. Users can also supply functions in place of single input values that will be evaluated with the parent player-key as an input and return the corresponding output value if it returns true. Similarly, the output values can also be functions that will transform the value of the parent player-key in some way. This creates a flexible method for that lets the user create both simple one-to-one relationships and more complex and dynamic mappings.

This method can be very useful for creating musical relationships between player object attributes that don't use numerical values. For example, instead of using a series of numbers to

```
p1 >> pluck([0, 2, 4, 6], dur=1)
p2 >> sitar(pitch + 2, amp=(pitch == 4))
```

Figure 6.9: FoxDot code using player-keys and a logical “equals to” test.

represent audio files stored on the computer, FoxDot uses a sequence of characters in a string¹. It would be impossible to use mathematical operators to create relationships with the non-numerical values in the string, but the `map` method provides a way to achieve this. For instance, the code in Figure 6.13 shows how the `map` method can be used to change the pitch of a player object based on the character being used to select a sample; when the character used by `p1` is “o” (which is a snare drum) the pitch for `p2` will be set to 2, and when the character is “=” (an open hi-hat), the pitch will hold the value of 4. Any other character will result in the pitch being 0, which is the default value if none of mapping functions are satisfied.

```
p1 >> pluck([0, 1, 2, 3, 4, 5])
p2 >> sitar(pan=(pitch >= 4) * -1)
```

Figure 6.10: FoxDot code using player-keys and a logical “greater than or equals to” test.

Being able to use functions as both input and output arguments in the mapping dictionary gives performers a flexible way of thinking about musical relationships between player objects. Without this feature a user could only map a single value in one attribute to another single output value. This helps solve issues such as users not knowing the range of values that might be used as input and mapping multiple input values to a single output value. Taking Figure 6.14 as an example; the pitch of `p2` will be 2 steps above that of `p1` when the pitch of `p1` has a value of less than, or equal to 4. If it is above 4 then the default option is used, which sets the pitch to 4 steps above the pitch held by `p1`. If the user has used single values to create a one-to-one mapping, then changes to `p1` would disrupt the mapping and leave the relationship meaningless as it would only be using the default values. Using logic in this way gives performers a flexibility and allows them to create versatile relationships that will still have meaning, even if changes are made to the code.

```
p1 >> pluck([0, 1, 2, 3, 4, 5])
p2 >> sitar(pan=(pitch >= 4) * -1 + (pitch < 2))
```

Figure 6.11: FoxDot code combining multiple logical tests with the same player-key.

¹The reason for doing so is discussed in Chapter 4, and similar practices are used in other live languages, such as *ixi-lang* (Magnusson, 2011b)

The use of functions in these mapping dictionaries starts to blur the line between object-oriented and functional programming in FoxDot. The ability to pass functions as input and outputs is similar to how other live coding languages operate, such as TidalCycles and Extempore, that are embedded in functional programming paradigms. I would argue, however, that the combination of functions in FoxDot is closer to the representation of musical behaviours and relationships than the building blocks of an algorithmic composition.

```
def map(self, mapping, default=0):

    # input functions
    functions = []

    # Convert default output to function
    if not callable(default):
        default_func = partial(lambda x, y: x, default)

    # Convert input values to functions
    for key, value in mapping.items():
        if not callable(key):
            test_func = partial(lambda x, y: x == y, key)

            if not callable(value):
                result_func = partial(lambda x, y: x, value)

            functions.append((test_func, result_func))

    # Define mapping function to test input functions
    def mapping_function(value):
        # For PGroups
        if isinstance(value, PGroup):
            new_values = []
            for item in value:
                new_values.append(mapping_function(item))
            return PGroup(new_values)

        # For other values
        for func, result in functions:
            if bool(func(value)) is True:
                return result(value)
        return default_func(value)

    return self.spawn_child(mapping_function)
```

Figure 6.12: Code for the player-key `map` method.

It is interesting to note that player objects can reference their own attributes to create useful and dynamic behaviours that can simplify processes that, otherwise, would require explicit hard-coding. For example, a player object can derive its stereo panning based its own pitch, such that higher-pitched notes are heard in the right channel, and lower-pitched notes in the left, as shown in Figure 6.15. In this trivial example, this does not offer a huge benefit to the coder but as the array of data representing pitch became more and more complex, the rules defined in the `map` method would still be applied consistently and the user would not have to worry about updating the panning attribute whenever they updated the pitch.

It is possible for a player-key to reference itself and create a recursive loop that would continue

```
p1 >> play("x-o(-([-o]=))", dur=1/2)
p2 >> pluck(p1.char.map({"o": 2, "=": 4}), dur=1/4)
```

Figure 6.13: FoxDot code using the player-key map method.

```
p1 >> pluck([0, 1, 2, 3, 4, 5])
p2 >> sitar(p1.pitch.map({lambda x: x <= 4: lambda y: y + 2}, default=lambda z: z + 4))
```

Figure 6.14: FoxDot code using functions as input for the player-key map method.

indefinitely, causing the FoxDot interpreter to crash. This can be caused by one player object referencing an attribute of another, which is subsequently updated to reference the attribute of the original player object. The player-key would continually look up its parent, which was itself, and then repeat this action indefinitely. Python does have a maximum number of recursive calls, and will raise an error if this is reached, but FoxDot will become unresponsive while this is happening. To combat this issue a test for circular referencing was added, which recursively checks a player-key's parent until it either finds the root or a reference to itself and raises an error in the latter case (shown in Figure 6.16).

6.3.2 Practice

In this section I reflect on several improvised performances by The Yorkshire Programming Ensemble using Troop in which we made use of the player-key class. One of these performances, Algorave Assembly, has already been discussed in some depth in the previous chapter, but in this section I will concentrate on the effect of using FoxDot, as opposed to Troop, on the ensemble interaction.

Together In Music conference, York - 14/04/18

Video recording: `ch6_2-Together_in_Music-14_04_18.mpg`.

See Appendix A.6 for performance description.

Together in Music was a three day conference held at the National Centre for Early Music in York that aimed to bring together academics and practitioners to examine ensemble performance from a variety of backgrounds and disciplines. I was part of the organising committee, along with PhD students, Sara D'Amario and Nicola Pennill, and academic staff from the universities of Leeds, Sheffield, and York. The conference was generously supported by both the White Rose College of the Arts and Humanities (WRoCAH) and the Society for Education, Music and Psychology Re-

```
p1 >> pluck([0, 1, 2, 3, 4, 5], pan=p1.pitch.map({lambda x: x > 2: -1}, default=1))
```

Figure 6.15: FoxDot code using player-key to map pitch to panning.

```
# Step 1
p1 >> pluck([1,2,3,4])
p2 >> pluck(dur=p1.pitch)

# Step 2
p1 >> pluck(p2.dur)

# Raises this error
ValueError: Circular reference found: p1.pitch to itself via p2.dur
```

Figure 6.16: Circular reference error.

search (SEMPRE). TYPE were asked to perform as part of the conference’s evening entertainment and showcase ensemble performance in a relatively unknown context; live coding. We had used the new player-key syntax several times in rehearsals leading up to this event and this was the first time we had used it in performance.

Algorave Assembly Lunchtime Concert, Leeds - 27/04/18

Video recording: `ch5.2-Algorave_Assembly-27.04.18.avi`.

See Section 5.4.2 and for more information about this performance and Appendix A.2 for its description.

6.3.3 Evaluation and outcomes

The Together in Music performance, like the Algorave Assembly lunchtime concert, was not governed by the aesthetic goal of making EDM. We approached this performance as a free improvisation although we were starting to develop our own style of experimental dance and psychedelic inspired music and knew we would be using that as a rough template for how the music would sound. When we weren’t playing at Algoraves, and we were afforded the opportunity, we tended to frame our performances as free improvisations guided by FoxDot’s sonic palette and metric constraints. The performances were much more focused on the process of creating the music than the music itself and achieving the feeling of “peak jamming”. We would often come into these performances with an initial idea and open minds. We started the performance by using soft drones and exploring the space in the sound and then just reacting to each other’s creative choices. This process of building on each other’s work and reacting to changes lead us into periods of various musical styles that involved funky percussive rhythms and psychedelic minimalist transcendence.

These are styles that exist within our repertoire but when and how they emerged in the performance were not discussed explicitly prior to the start of the performance. Part of improvisation and experimentation also means that a small mistake can have large consequences on the music. For example, at 17:30 Laurie changes a synth's amplitude to 869 instead of .869, which creates a distorted growl that dominates anything else in the mix for over ten seconds. These 'errors' are an important part of the live coding aesthetic for me, though, as it highlights that these performances are real and are being created live for the audience by humans, not computers. We don't always know what is about to happen next and if we tell the computer to do something stupid, it won't think twice about doing it.

We also made a conscious effort to use more player-key syntax where possible to try and create a more cohesive ensemble performance whose musical elements felt connected and with purpose. We used the player-keys to great effect but Lucy and Laurie both subverted my expectations by exploring their unintended uses as well. At minute 4 Lucy mapped the duration of one synth to the "chop" effect of another using the code `chop = z1.dur`. This was interesting for several reasons; The first is that the player-key was being used as an input value for an attribute that it itself did not represent, i.e. it was not being used as a duration value. The second is that it is an effective way to explore *un-intention* in improvised live coding performance. There is often an explicitness to the way we, as an ensemble, write our code but using player-keys in this way introduces a level of uncertainty, which makes the process rather exciting. In the context of the performance itself, it created an echoing polyrhythmic effect, as the chopped up elements of the synth overlapped themselves in unpredictable rhythms. Laurie also explored non-direct mappings by setting the amplitude of one player follow the values used in the "echo" effect applied to a percussive sequence using `amp = p3.echo`. The "echo" value was being randomly selected from 0, 0.25, 0.5, or 0.75 and this caused the synth to drop in out of the mix as the amount of echo varied. Like Lucy, Laurie was not using the player-key to copy an attribute of one player object to another, but explored the unintended consequences of creating a musical relationship between two layers across their different attributes. We try to keep an element of experimentation in our improvisation and subverting the computer program's intended use creates unexpected outcomes that keep performances fresh.

Just as it was for the the `follow` method, the player-key objects were useful as a platform for getting started in the performance. It allowed us as an ensemble to work on separate code but use a single connected thread to tie it together musically. Player-keys were also more versatile than the `follow` method, as they allowed us to utilise attributes of the player objects other than pitch. For example, in the Together In Music performance, Lucy connected two player objects via their low-pass filter values to ensure that their prominence within the overall mix would be equal.

The two musical layers shared the same values for the same attribute and the relationship could be easily identified within the music, which could be described as a ‘direct’ musical relationship created by the player-key. There were also several instances of ‘non-direct’ musical relationships created using player-keys where one player object’s attribute is used as an input for a different attribute. An example of this can be taken from the same performance when Laurie uses the syntax `amp = p3.echo` to connect the amplitude of a melodic synth to the amount of the “echo” effect (a comb delay) was applied to a percussive sequence. The synth would only have a non-silent amplitude when the echo effect was being applied and this created quite stark contrasts in density in the music, which would range from scattered strikes of drum kit samples to densely packed percussive reverberations combined with warm electronic synth tones.

The player-key class has definitely provided us with more options for ensemble communication within the code than just using the `follow` method. We are not limited into only thinking about pitch when creating musical relationships, but can also consider rhythm, amplitude, and timbre, which made for many interesting musical moments during the performances. Like the `follow` method, however, there were also some unexpected uses of the player-key that resulted in the development of novel musical ideas or acted as catalysts for musical change. For example, Lucy doubling and halving pitch values in both performances was not something I would have thought to do myself, but worked well in both cases. Halfway through the Together in Music performance Lucy doubled the pitch values in the player object, 12, which was using the “prophet” synth and taking its pitch from a fast-moving melody generated by another player object, z4. Due to the large difference in durations of the two player objects, the pitch being picked up by 12 was always near the start of the melody’s cycle and varied little. Multiplying the value by two, however, increased the variation in tonality and moved it into a higher register, which helped generate momentum as we began working on new musical ideas. During the Algorave Assembly performance, Lucy also halved the value of a player-key to use at pitch data. This would have had a dramatic affect on tonality had any of the source player-key’s values been odd and caused Lucy’s synth to play sharp notes. Looking back I think I don’t think that would have worked too well in the context of the performance at that time but it would have been interesting to see how we adapted to the discordant texture of the music that would surely have emerged.

While we had experimented with using it in rehearsal, at no point in these performances did we make use of the `map` method. This was disappointing as I felt it provided a means to create more complex musical relationship and could make for interesting conditional structures as part of the algorithmic composition aspect of live coding. However, after discussing this with Laurie and Lucy, we all agreed that part of the problem was that the syntax was quite complicated and required a deeper knowledge of the Python language, and functional programming as a whole, to use to its full potential. I have used it on a few other occasions during a performance, but,

due to the complexity of the input arguments and lack of syntax highlighting in Troop, the code was never touched by the other members of the ensemble. As a domain-specific language, FoxDot only requires the user to know a handful of Python commands before being able to create music. However, going beyond its fundamentals seemingly demanded users learn more about computer programming than music making, which did not happen as organically during rehearsals as it had done for the `follow` method and the basic player-key syntax.

It is interesting to see how much functional programming was present in the development of the player-key class given that a choice was made at the start of this chapter to use an object-oriented programming model instead. Perhaps it is only possible to represent real-world ensemble behaviour in code through the combination of multiple programming paradigms. The use of inline lambda functions in Python is quite a verbose process and the `map` method would often combine multiple instances of these when used. Those unfamiliar with Python's `lambda` keyword, such as Laurie and Lucy, may find the use of `map` daunting because of this. To further complicate the matter, the `map` method also requires users to create a Python dictionary of mappings that include lambda functions, which could also be difficult to newer users of the language.

I would still like to create more interesting musical relationships that are more complex than simple logical tests or mathematical operations, but the execution would need to be simpler than what has been developed so far. One of the aims of this PhD is to lower the boundaries to entry to ensemble live coding and the `map` method does not do this; even Laurie and Lucy, who have been using FoxDot for over a year, were put off by the complicated syntax and use of lambda functions. The `map` method is also difficult to read from the audience's perspective. It is not a requirement for performers' code to be easily interpreted by the audience, and it very often isn't, but it is my own opinion that by making code easier to interpret for the layperson, live coding can become a more open and inclusive practice.

6.4 Phase 3: Extending Player-Keys for Musical Behaviours

6.4.1 Development

So far the examples using the player-Key data structure have only implemented explicit relationships, such that given a value for the parent player-key the value of the child will be predetermined. This can be useful in replicating some existing musical techniques but, across the many genres of music that exist, relationships are not always so straightforward. In fact, what is being described as dynamic relationships may not be *relationships* but *behaviours*, as Emmerson (2007) suggests:

Social *behaviour* implies a process in time, the dynamic interaction of elements, while social *relationship* may be a static snapshot of a state of affairs (possibly from within

a behaviour sequence) and is thus independent of narrative time. This is the deepest and longest established of music’s many models (p. 45)

The direct relationship created through simple mathematical functions, such as `p1.pitch + 4`, is actually just a static snapshot of a greater and more complex relationship that changes over time, which Emmerson considers as musical behaviour. Up to this point, the transformation of player-keys have only been performed using standard mathematical operators that are part of the Python programming language. This does mean that player-key relationships can be created easily and their meaning easily inferred, but they are inflexible for the end user; input values supplied to create player-key relationships must be a number or another player-key. What happens if a user wishes to create a relationship that involves calling an external function, for example? They are unable to do so as they are only limited to the basic arithmetic operations. To combat this, a `transform` method has been added to the player-key class to allow users to supply their own functions and customise their player-key relationships. For example, if a user wishes the combine the current beat with a player-key using modulo division to derive a value, they may think that using `Clock.now() % p1.pitch` would be the correct way to achieve this. However, `Clock.now()` would return only a single number that represents the time in the clock at the point at which the line code was evaluated, which would not increase with time. Figure 6.17 shows how the `transform` method can be used to correctly combine the current beat number with a player-key using a simple lambda function that calls `Clock.now()` every time a new value is requested from the player-key.

```
# Basic code to start a player object
p1 >> pluck([1, 2, 3, 4])

# Incorrect way of relating p1.pitch the current beat
p2 >> pads(Clock.now() % p1.pitch)

# Correct way of relating p1.pitch the current beat
p2 >> pads(p1.pitch.transform(lambda x: Clock.now() % x))
```

Figure 6.17: FoxDot code using the player-key `transform` method.

This is a trivial example but demonstrates how users can incorporate their own custom functions to create more dynamic player-key relationships that may evolve over time. However, this does require the use of Python lambda functions, which, as discussed in Section 6.3.3, are often tricky to use. The `transform` method can take any “callable” object as an input, which means users can supply both functions (as above) and any object with a valid `__call__` method. Adding the `transform` method enables a Python object to be called as if it were a function but allows for the object’s state to be altered. Combining the player-key’s `transform` method with a callable object that holds state – and can use that state to make decisions about what values to return – can help

create more interesting (and perhaps indeterminate) behaviour for the player-key. Some real-world musical behaviours can only be represented through this method, such as pitch accompaniment.

```
# Basic code to start a player object
p1 >> pluck([1, 2, 3, 4], dur=8)

# Using 'accompany()' transforms the player-key
p2 >> pads(p1.pitch.accompany())
```

Figure 6.18: FoxDot code using the the player-key `accompany` method.

An `Accompany` class was created to represent the behaviour of pitch accompaniment in a way that can be used with the player-key’s `transform` method. When calculating its own value, a player-key will call its given `Accompany` object with itself as an argument. The `Accompany` object will then check if the parent key’s value has changed since the last call and return a new value if so. The new value will be the closest note that completes a third or fifth above or below the parent key’s value. To introduce some indeterminacy into the procedure, some randomness was also introduced to select the second closest harmonic value one third of the time and the third closest value one tenth of the time.

To represent this behaviour mathematically, we will let x be the parent player-key and let y be the player-key that is accompanying x . We can derive the player-key’s current value (y_t) by selecting the value closest to y_{t-1} from the set $[x_t - 5, x_t - 3, x_t, x_t + 2, x_t + 4]$. This simple function creates a dynamic behaviour and ties together two musical strands to create organic sounding pitch material. To help simplify its use during performance, an `accompany` method was added to the player-key class so users don’t have to invoke the `Accompany` class directly using `transform`, as shown in Figure 6.18. Of course, accompaniment is not always just completing a third or a fifth above a note and the relationship can be configured by the user by supplying different values when calling the `accompany` method.

Another type of ensemble behaviour that is of interest is that of “play” within indeterminate music; processes that occur during a performance in which “[t]here may be some autonomy, with players exercising choice about how they might realize a piece based on internal preferences, or possibly a self-determined aim or purpose” (Saunders, 2017). In ensemble-based pieces this is sometimes accompanied by a goal, such as reaching a group consensus like in the piece *reaching an acceptable and stable solution* (2018) by James Saunders, but indeterminate music will always have a form that is defined by rules for what actions can and cannot be taken by performers. This can range from performers being given almost complete free-reign over the music, as in *Musicircus* (1967) by John Cage, through to the rigorously structured rules, like in the game piece, *Cobra* (1984), by John Zorn.

```

def versus(self, other_key, rule=lambda x, y: x > y, attr=None):
    """ Sets the 'amplify' key for both players to
        be dependent on the comparison of keys """

    # Get reference to the second player object

    other = other_key.player

    # Get the attribute from the key to versus

    this_key = getattr(self, other_key.attr if attr is None else attr)

    # Set amplifications based on the rule

    self.amplify = this_key.transform(lambda x: rule(x, other_key.now()))
    other.amplify = this_key.transform(lambda x: not rule(x, other_key.now()))

    return self

```

Figure 6.19: Code for the `versus` method from the player object class.

Live coding subverts the idea of “play” by having the human performer define the rules for performance and then re-define them while they are being followed by the computer. This could be simply writing notation and changing it while it is being played or implementing a stochastic probability model and tweaking its input parameters during the performance. At the heart of indeterminacy lies improvisation, which is integral to the human processes of live coding. Considering the computer as a performer, though, a live coding performance is only indeterminate because the rules (algorithms) are not known prior to the start. Without creating software for writing its own code, such as Tidal Autocode², the computer is given little autonomy within a performance. Can the computer be given the agency to “play” within a live coding performance, and can that process be propagated from the computer to the human performers through representation of “play” in the code?

There is an entire field of research into computer simulation of musical creativity and the application of machine learning in music but it is beyond the scope of this PhD to explore these concepts in ensemble live coding; instead of giving the computer the ability to make informed creative decisions mid-performance, playful rules are being explored that consider the output of each performer’s code to create indeterminate musical structures. “Sometimes the beauty of play resides in the tension between control and chaos” (Sicart, 2014, p. 83) and even simple rules that determine when player objects can and cannot play could give them the agency to disrupt the performance in some unexpected way.

As opposed to adding the functionality of the behaviour to the player-key itself, a method has been added to the player object class, called `versus`. It takes a player-key as an input argument and, by default, compares its value to that of the same player-key attribute of the player object. It

²<https://github.com/kindohm/tidal-autocode>, accessed: 05/04/19

```

# Basic code to start a player object
p1 >> pluck([1, 2, 3, 4], pan=[-1, -0.5, 0.5, 1])

# Start a "versus" against the pitch value in p1
p2 >> pads([0, 5, 2]).versus(p1.pitch)

# Scale and map
p2 >> pads([0, 5, 2], dur=1/2).versus(abs(p1.pan) * 4, attr="pitch")

```

Figure 6.20: FoxDot code using the `versus` method.

then maps the output of this comparison to an “amplify” attribute that acts as a logic gate for playing a note. By default the comparison only tests which value is larger but this can be configured by the user by supplying a lambda function. For example, if a user supplies a pitch player-key as the input, then method creates a relationship between two player objects where only the one with the highest pitch will play a note at any one time. A user can also compare two different attributes by specifying the second attribute as an argument, as shown in Figure 6.20. Different attributes often operate within different ranges but can be scaled accordingly by multiplying the player-key input by an appropriate factor. For example, values for panning will always be between -1 and 1 whereas pitch can be almost any number. As shown in Figure 6.20, multiplying the absolute value of the panning player-key by 4 will scale the range from -1 to 1 to between 0 and 4, making it much more suitable for comparison to pitch values.

6.4.2 Practice

Algo-Rhythms, Rotterdam, 28/04/2019

Video recording: `ch6_3-Algo_Rhythms-28.04.19.mpg`.

See Appendix A.7 for performance description.

This event defined itself as “deconstructing the way we think about the myriad of socio-cultural possibilities that the interface between music and digital technology opens up”³ and took place at a venue called Worm Rotterdam but, due to some scheduling conflicts, only Lucy and Laurie were able to attend the performance in person and Innocent and I had to connect remotely over the internet. The difference in performing remotely compared to live in front of an audience is stark. Without being able to feel the energy in a room and feed off of it during the performance, you often have to rely on information from people in the venue. It’s a very strange feeling; listening to your performance through your headphones gives you no real indication as to what the sounds are actually like in the room’s acoustics and the level of anxiety before a performance is considerably

³<https://worm.org/production/algo-rhythms-3/>, accessed: 08/05/2019

less. Even though I was the only one on the stage during the ICLI performance, discussed in Section 5.5.2, it felt much more of a group performance than this one. Perhaps this is because that the ICLI performance met more of my expectations of what a performance should be; a stage, a sound system, an audience. Most of our communication takes place within the code so the fact that Laurie and Lucy were performing remotely didn't actually affect me. However, being the remote performer was much more discomfoting and was perhaps the stimulus for some of the more awkward rhythmic sections that occurred, particularly at the start of the performance. It is interesting to see that technology can do so much to connect us in music over great distances but it cannot replace a live audience.

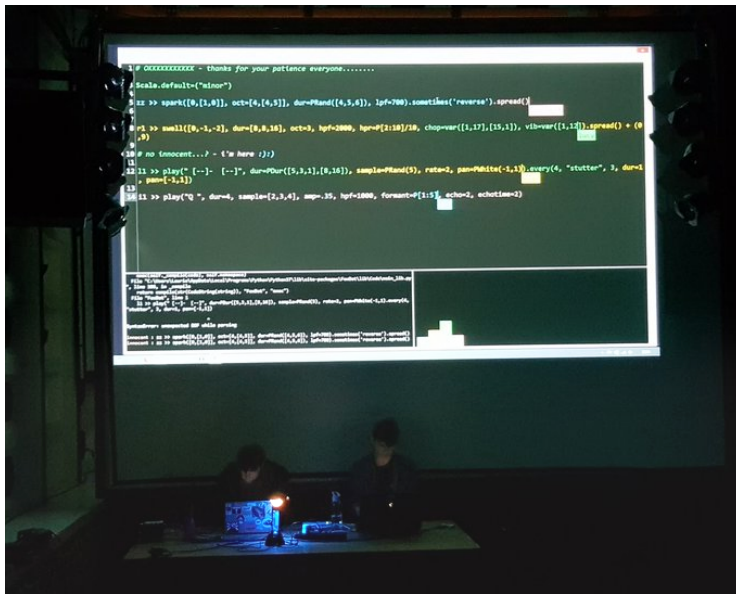


Figure 6.21: Photo from Algo-Rhythms performance. Courtesy of Creative Coding Amsterdam.

6.4.3 Evaluation and outcomes

Before we started the performance, Lucy texted the group to say that the event was “more of a sit down kind of thing” so we decided not to make dance music, but to focus more on the texture of the sounds we were making. This posed an interesting challenge as FoxDot is more tailored to creating rhythms and melodies, but it does afford a degree of control over sound quality in certain contexts. It also posed a problem when considering the nature of the new functionality that has been added into FoxDot during this development phase; melodic accompaniment and playful boolean comparisons for switching musical layers “on” and “off”. The latter looks at FoxDot’s musical components as discrete musical events but timbre-based music very often does not. Similarly, if a piece of music is not focused on the combination of harmonic elements, then the implementation of melodic accompaniment may not even be required. That being said, the `accompany` method was used at several points during the performance but the `versus` method was

only used once.

It was clear after the performance that, even though I felt I had demonstrated how to use the `versus` method to the rest of the group, they did not truly understand how it worked or how to use it. In its most simple implementation, it compares pitch values of two FoxDot player objects, but pitch was not always central to this performance and using `versus` with other FoxDot attributes (and scaling values) requires several extra keyword arguments. The `accompany` method, on the other hand, does not require extra keyword arguments and is only really applicable to the pitch attributes. Even though we did not make tonality a focal point of the performance, we could easily add a melodic accompaniment that would add a generative element to the music. An example of this was around minute 22 where Lucy used `zc.pitch.accompany()` to connect a marimba synth to the “donk” layer that was playing microtonal pitch values. I hadn’t explored the idea of accompaniment in microtonality but the result of this usage meant that the two musical sequences seemed to “fit” together without creating much dissonance. It also meant that Lucy did not have to think about microtonal values when setting this up as it would all be handled by FoxDot. Another unexpected use of the `accompany` method occurred halfway through the fifth minute when Innocent used `r1.pitch.accompany()` in a sliding “prophet” layer, `i2`. At that point the `r1` layer was playing two notes together, both of which were “accompanied” by the `i2` layer, adding a level of intensity to the sound and also creating a connection between the musical elements. I think, in some respects, the `accompany` method was used as “play” as a way of exploring unintended consequences and leaving some creative decisions up to the live coding language and then pursuing the musical output that follows.

The infrequent use of the `versus` method suggests that learning new functions and syntax for live coding takes more than just one or two practices and needs to be embedded into a personal repertoire before being utilised during performance. At the time of writing TYPE had been performing together for two years and we had developed a certain style of live coding, which makes it quite difficult to accommodate new language-based features. In some instances it has been straightforward because the new syntax felt like a natural progression from the older syntax, such as when using the `accompany` method. Using player attributes via the `player-key` class has been prevalent in our practice for some time now and adding a single, self-descriptive method was very easy to incorporate into performance. The `versus` method, however, uses a different structure to any function we had used before; the method is appended to the player object’s `SynthDef` and takes a `player-key` reference as an input, which is unlike any syntax we would use, which is probably why the ensemble was so reluctant to try it out during the live performance.

Another issue with the performance was the network connectivity problem that occurred during the performance where we had to start again. Ordinarily if someone is disconnected from Troop due to networking issues, they can disconnect and log in again. Unfortunately Lucy had issues

reconnecting as she was never properly disconnected from the server and her laptop was also responsible for generating audio for the audience in Rotterdam, so troubleshooting was not feasible for her. The best course of action was to start over completely and we were able to finish the performance without any more interruptions. However, it did take a few minutes to find the group flow again after we had resumed. We were peak jamming at the point of the crash and suddenly it was stopped against our will, which brought on a very frustrating feeling. Once you are out of that flow, it is very hard to find it again, especially given a short time constraint, which adds pressure to the situation. We managed to find a middle ground by the end but the music immediately after the break was particularly weak because of this break in flow.

6.5 Conclusions

6.5.1 Personal reflection

Since the start of this PhD research, FoxDot has grown significantly as a language and its breadth of functionality with relation to ensemble performance has increased with it. Several of these additions have become frequently used tools within the improvisational work of TYPE and have proven beneficial to our practice. FoxDot has helped improve our ability to succinctly express collaborative musical ideas and foster creativity during performance. For example, during phase 1 of development, we often used the `follow` at the start of a jam to connect our lines of code musically, which allowed us to focus on other aspects, such as rhythm or timbre. The nature of the `follow` meant that two lines would share the same base pitch value regardless of any changes made to either line of code, thus allowing performers to edit their musical layer without the worrying about playing unsuitable notes.

The use of the `player-key` data type in phase 2 was very similar. It was often used early in performances, enabling members of the ensemble to connect to a single musical thread, such as one sequence's pitch, but work on other aspects of the music independently. It also gave users more flexibility in what attribute their code could follow. For example, when Lucy used the low-pass filter `player-key` as an input for another layer's low-pass filter during the Together In Music performance, she brought the two sequences together in terms of their equalisation, which is not something that the `follow` method would have enabled her to do. The `player-key` also enabled more flexible pitch-based relationships too. Using `follow` on a `player` object playing multiple notes together in a chord would copy all of the notes being played but, by using a `player-key`'s `_getitem_` method, users can specify which note of the chord they wish to access. This helped create multi-directional musical relationships within the code where, previously, they were only uni-directional (in the sense that multi-note sequences could only be added to a single note sequence

but now a single note sequence can be determined by a multi-note sequence). The player-key was also integral to one of my personal favourite moments during a TYPE performance. During the Algorave Assembly performance, Laurie created shifting chord accompaniment at minute 21 by using the syntax `b1.degree + var([2,4,6])`, which complimented the bassline I was creating perfectly. This was a very satisfying moment and was facilitated by the very simple feature of the language.

There was also a lot of experimentation among the group with the player-key class. For example, at minute 15:35 during the Together in Music performance, Lucy uses the syntax `z4.degree * 2` to double the pitch degree of a player-key and create an exponential relationship between the pitches of two musical sequences. Where doubling the frequency would play the same pitch in a higher octave, this effect increased the range of pitches played, which created a more drastic melody that juxtaposed with its source. Similarly, during the Algorave Assembly performance, Lucy divided a pitch player-key by two, which created a relationship in terms of movement up and down the scale, but was not explicitly one-to-one and made for some interesting tonal combinations. Another interesting use of the player-key data-type occurred when performers created relationships between different musical elements, such as when Laurie linked a layer's echo delay time to another layer's amplitude during the Together In Music performance. The values were both appropriately between 0 and 1 and, when implemented, created the effect of two sounds increasing and decreasing in density together as one would be louder and the other repeated and embellished through its echo effect. As mentioned in Section 6.3.2, this practice was an effective way of exploring unintended consequences in live performances. Using the same attribute for sharing player-keys, i.e. using the syntax `dur = p1.dur`, would give the user an expected result. They are already aware of the duration of `p1` and are essentially reusing it in another facet of the music. Using different attributes, however, introduces an element of uncertainty and indeterminacy.

While some features introduced to FoxDot were beneficial for mediating ensemble interaction, some were perhaps too complicated to be fully realised during performance. I thought that the `map` method developed in phase 2 would be a powerful tool that would enable the group to create interesting musical relationships but it was rarely used during performance. The use of Python's `lambda` syntax with this feature seemed to be too complex compared to just combining player-keys with simple mathematical operators. I have found it very difficult to make musical transformations flexible while still remaining simple to use. FoxDot was designed to be a simple musical interface and that is how we, as a group, learned to use it; simply. Adding levels of complexity that require deeper understanding of Python's functional programming modules was a step out of our comfort zone, which made it hard to include in our practice. The same was true for the `versus` method, which was not utilised by any other member of the ensemble (other than myself) due to its dissimilarity to any other syntax that we tended use in performance. This is an example of

Norman’s warning that “designers are not typical users” (Norman, 1998, p. 151) in that I had become so expert in the system I was unable to foresee that anyone else would have any problems using it. However, testing these features with a group of end-users has made these issues clear and helped guide development in a more user-centred direction.

The FoxDot functions introduced in this chapter that had similarities to natural language, such as `follow`, basic player-key syntax like `p1.pitch`, and `accompany`, were successfully (and frequently) used by the ensemble during performances. Other, more complex, features were not incorporated into our practice as easily and this can most likely be attributed to the time taken to learn them. Children are often taught programming with the use of drag-and-drop graphical interfaces, such as *Scratch* (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), because representations of physical objects are easier to understand than code that is delimited by unusual characters such as semicolons and curly braces, which is one of the reasons why live coding is such a fundamentally challenging practice. Brusilovsky, Calabrese, Hvorecky, Kouchnirenko, and Miller (1997) suggest that “mini-languages” are better alternatives to learning general-purpose languages. They state that the large size of a general-purpose language “makes it difficult to understand the material properly, thereby failing to form a strong cognitive infrastructure” when learning it (p. 67). FoxDot can be considered a mini-language embedded within the Python general purpose language. As a wider range of syntax from Python was introduced throughout this chapter, such as `lambda` functions, the more difficult it became for members of TYPE to implement the functionality they necessitated. FoxDot’s role as a mini-language began to blur with Python’s extensive general-purpose library and the less experienced programmers struggled during live performance. Using more common aspects of natural language might be able to help audiences relate the code they see to the outcome they hear, which is important to the live coding audience experience (Burland & McLean, 2016).

Given that I started this chapter by discussing the benefits of OOP for developing the player-key functionality I was surprised by the amount of functional programming that was required in the software development. Although it was not often used during performance, the use of `lambda` functions was necessary in the back-end code for defining the player-key behaviours. Whenever a child player-key was created, it required passing it information about the parent and a transformation function. Computer music has a history of focusing on the transformation of musical structures, such as Laurie Spiegel’s *Manipulation of Musical Patterns* (Spiegel, 1981), and functional programming provides a means for representing these transformations as well as combining them and moving them around a program as data themselves. However, functional programming is not a widely taught programming paradigm, even at a university level, and the concept shares more commonalities with high-level mathematics than it does with natural language. This means it can be difficult for many new programmers and computer musicians to develop an effective un-

derstanding of functional programming such that they would be able to use it in their own practice. There is an unfortunate trade-off between the power and flexibility of functional programming as a creative interface and the semiotic familiarity that can be made available through simple OOP design.

The foremost research question that is being addressed in this PhD is “how can collaboration in ensemble live coding be better facilitated through performance systems and interface design?”. A live coding language is a performance system in its own right; it is a technology that enables musical creation in a live setting. Adapting the language to help facilitate inter-personal collaboration in live coding ensembles has been successful, to a degree, and has also opened new avenues for exploring indeterminacy in our performances.

6.5.2 User evaluation

Once again the ideas of “trust”, “flow”, and “immediacy” with respect to the collaborative creative interface were discussed in an interview with all members of TYPE, but this time with regards to FoxDot itself as opposed to Troop. The success or failure of each collaborative element of functionality that was added to FoxDot was also discussed as part of the interview. There was unanimous praise for FoxDot’s clear syntax, which made it easy to collaborate, especially when editing the same line of code as a co-performer. However, this is related more to Python’s focus on readability as opposed to any FoxDot functionality developed during this chapter and will not be discussed here. After outlining the different collaborative functions that were added to FoxDot, i.e. the `follow` method, `player-keys`, and extended `player-key` methods, the group was asked for their general thoughts on the language as a facilitator of ensemble improvisation. Lucy stated:

It is nice because it’s really easy to, like, drop something in that is coherent with something somebody else has done. You can borrow, say, like, the pitch values, erm or use, like, accompany or follow, which I’d completely forgotten about, as a way of making sure what you’re doing is in line and not clashing.

It was the general consensus that, while useful at the time, the `follow` method had been superseded by the `play-key` syntax as the way to share pitch information among FoxDot players while playing. It was also noted that using the `player-key` structures helped manage the amount of code being written in the Troop text buffer and better keep track of what other user’s were doing.

Lucy: “I liked it [the `player-key` syntax] because I think like, well, we tried out defining patterns didn’t we? And that was good but it just felt like quite a lot to kind of get your head round whereas, like the, using the `p1.pitch` or `p1.amp` or whatever seemed a much, like, speedier way of taking, like, little bits of pattern from other parts of the

code and reusing them. I don't know why, maybe it's like another thing to think about if there's another line of code that's got the pattern in it somehow."

Laurie: "I think if you set up too many patterns, melodic or rhythmic patterns as well, often it can get a bit confusing [...] it kind of unifies, the pitch especially, a bit more than having it all over the place and less musical definition maybe. That might be even more of a problem when you got so many people that are working at the same time. It can become a bit overloaded tonally."

Lucy: "Yeah I think, definitely since we've become a four piece, we've had to keep a bit more of an eye on the expansion of the code [everyone laughs] because it was already a bit of a problem when there was three of us. And now there's four of us, it's like... [hand gestures] and I think that does, as Laurie says, it simplifies all of that"

Ryan: "The player-keys?"

Lucy: "Yeah because you're not, like..."

Innocent: "It makes it easier to keep everything in line so that we don't have clashing sounds. Umm, if someone's creating, umm, a different amp pattern and someone has a different amp pattern over there, but if you unify those and then change one thing slightly, umm, then it's more in line with everything else. Umm, which means you're collaborating... the sound is more unified than when everything is playing individually, so it's more of a song. It's more musical."

Lucy: "I think it helps the transitions as well between, like, sections because you can transition, like, so you've got [...] four players all referencing the same, say, pitch pattern. You can start to change that pattern and it transitions, you know, the music in a much more, kind of, consistent and coherent way."

The player-key syntax not only makes it easier to manage the amount of code being used by the ensemble, it also helps manage tonality and transitions between musical sections. For Innocent, the idea of achieving more musicality in performances is important and player-keys allow us to do this easily as a group. Of course, arhythmic and atonal music can be more musically interesting for many listeners but this is arguably easier to create as an improvising ensemble (as opposed to more rhythmically and harmonically complimentary music) as you can just disregard your co-performer's code. When asked about the level of trust in FoxDot with the ability to take more risks, Lucy responded:

I don't know because I wouldn't necessarily do anything particularly risky with it but I think it improves your confidence of like 'Oh I'm gonna bring in a new player'

and I know it's gonna sound alright because I'm gonna borrow the pitch values from somewhere else for example

This sentiment was shared by the rest of the group; the player-key functionality may not increase the level of risk-taking in an action, but gives performers more confidence in completing tasks, such as adding a new line of code. Sharing musical information, such as pitch or rhythm, in this simple way allows for new code to be introduced into a performance with the confidence it will not be “clashing” and can then be subjected to the iterative process of editing and evaluation. Like using the `follow` method as a jumping off point for musical ideas, this was not a technique that was explicitly discussed and developed by the group, but naturally emerged through practice. It is another example of tacit knowledge being created through real-world use of the software over a period of time.

The group was then asked about why some functions, such as the player-key `map`, `transform`, and `versus` methods were never really utilised in live performances even though they were introduced and explained in rehearsals. Below was the response:

Lucy: “I think in terms of technical knowledge it was just a bit over my head [...] I noticed something that when I was playing with Graham [Dunning] in London actually [...] I noticed I was using much simpler code than I would solo and when I was trying to do something complex I was, like, bobbing my head enjoying myself and then I would, like, try and do a complicated bit of a code and I would, like, stop and kind of back off and it was like taking me out of it a little bit, like out of the flow of the performance [...] I actually do know that stuff but it's almost like, to stop and think about it almost like breaks me out of that like improvisational flow. And so I tend to just stick with like the simpler bits”

Innocent: “I think if it's more complicated, if the syntax is more complicated, especially in a live environment having to think about that takes you out of the flow”

Laurie: “It becomes more of a technical exercise than expressing yourself”

It is clear that the gap in technical knowledge of FoxDot, and functional programming within Python, between myself and the rest of the group was larger than I had anticipated. The cognitive load put on less experienced Python programmers to incorporate more powerful, yet complicated, functions was more than enough to affect their improvisational flow. As mentioned previously, the achievement of flow is an important part of group improvisation and disrupting that can be detrimental to performance. It is interesting to note, however, that Lucy had used Troop to perform using TidalCycles and found that, even though she was confident with her technical abilities in that language and would utilise them during solo performance, she found herself using much simpler

techniques to avoid breaking any group flow with her collaborator. She elaborates on why she feels using less complicated code in ensemble performance is important in achieving flow as part of a group:

I think [using simple code] is more important when you're playing in a group because you need to be much more tuned in to what everyone else is doing whereas if I'm like playing solo [...] I will, like, have a much better handle on what's going on because I'm in control of everything and so I'm more likely to do that step back, put in a complicated bit of syntax whereas I think with FoxDot, and particularly when jamming with Troop, I'd say [...] I'm so, like, conscious of what you guys are all doing and typing at pretty much all times, like, with my eyes darting about, I almost don't have the headroom to be, like, 'Oh I'm gonna do a lambda function that I don't really know how to do'. Like, it's just kind of more... the energy works a bit better if it's simple I think [...] and you can still do so much that's like really great sounding with the simple syntax. I don't feel limited, really, by the kind of, well, simpler functions.

The idea that the “energy works a bit better if it's simple” is particularly interesting. Simple code means that everyone is able to understand it and will not not require any extra time or cognitive energy to do so. The downside of reducing the technical complexity of the syntax is that it arguably lowers the perceived skill levels of the performers in that they are not demonstrating to the audience a deep technical understanding that might be expected of a virtuoso coder. However, the creative utility of the performers has been demonstrated several times in our practice through the re-purposing of these tools in search of unintended consequences and outputs. For example, when Laurie mapped a the value of an “echo” effect to a player object's amplitude, he created a novel musical relationship using very simple syntax that transcended the player-key's original design intentions. Furthermore, with simple syntax, there is no exclusion within the group between those who can understand and those who cannot:

Laurie: “It's like, being able to share it [the complex syntax] amongst everyone... everyone would need to be really solid with it, and [if] that's not the case – [or] it's less likely to be the case – then it's less collaborative I guess.”

Lucy: “Yeah sometimes I see you doing stuff, Ryan, and I'm like ‘what the fuck?’ [laughs] and I don't want to go and, like, change values because I like literally would be like-”

Innocent: “Yeah you don't want to break it.”

Lucy: “I think that's not necessarily a problem but I do think for like a truly collaborative performance you are kind of a little bit bound by that simplicity, because I notice if you write a really complicated line of code then the rest of us don't touch it.”

The imbalance in code complexity between myself and the rest of the ensemble can lead to moments where very little collaboration occurs due to members of the ensemble not having the confidence in their knowledge to edit the code. One of the research questions posited in this PhD is “how can collaboration in ensemble live coding better facilitated through performance systems, such as language, and interface design?” and it seems that making syntax for collaboration as simple as possible is a key aspect of designing a language for collaborative live coding. The complexity of the syntax also affected the immediacy that the group felt when performing:

Innocent: “The limitation comes from your level of knowledge. If you don’t know the syntax to take the idea from your head onto a screen then that’s where the limitation is. That’s where you don’t get that immediacy because you’re taking the time to think about how to translate.”

Lucy: “Exactly! It’s that translation, exactly that.”

Innocent: “So if you know more, if you have more technical knowledge about, umm, sort of the language that we use then you can just do that a lot easier”

Lucy: “Without thinking about it almost [...] I just do it. But like, that’s the difference, it’s that fluency of, like, ‘am I translating or do I just know’ [...] there’s an element of, like, translation, that’s exactly it, I hadn’t thought about it that way.”

Laurie: “I think what you were saying earlier as well about a lot of the changes that people were making were incremental. They’re not, kind of- a lot of them aren’t huge, they’re all kind of small little... you can actually imagine the sound you’re hearing with that one change that you make.”

Innocent: “You start with something small and then you might hear a change you want to make and then you make that one change and then it’s like, yeah-”

Lucy: “Yeah it’s tweaking isn’t it? Whereas If you’re like suddenly like ‘I wanna take the pitch from that and do this to it and blah blah blah’ It gets [to] a point of complexity where you can’t hold that in you head along with all the other stuff you’re doing.”

The sense of immediacy felt by the group when collaborating over FoxDot comes with a deeper technical understanding of the language, which itself, develops with time and practice. Lucy was also asked to compare her performance with Troop using Tidal with those performing with TYPE and FoxDot. Below is her reply:

There’s two things. So one is Tidal just doesn’t have those nice ways of, like, interacting between two different lines of code or two different players so instantly that’s functionality that’s just not there with Tidal and, erm, for all the reasons we discussed earlier, that’s a really positive thing about FoxDot. The other thing, I would say, which

is probably like a bit less tangible is that with FoxDot I've only really played in TYPE and so I've developed my whole style of coding around you guys. Now I've played collaborative gigs with Tidal – I've played with Alex [McLean] and with Graham [Dunning] – and they both have completely different styles and completely different from my style of coding and that does make it quite difficult to, like, sometimes interact with each other's code because they'll be, like, using a function that I never use and don't really understand and don't really know how it fits together and they'll be using, like, different samples or you know just a completely different approach. I think it's [FoxDot] much better set up for that [collaboration] and I think you've made conscious decisions around that in the development of it and I think that definitely plays out when we're jamming together. And I would say compared to when we first started as to now, I feel like it's just got better and better and again that's a little bit about our development as users but also the development of the software

The focus on collaborative functionality in FoxDot has improved Lucy's experience of ensemble performance but it was not the only thing to do so. It was also benefited by the time spent rehearsing together, developing tacit knowledge of the systems, and a style of coding that revolves around collaboration.

6.5.3 Final thoughts

While not all of the features developed over the course of this chapter were fully utilised in TYPE's creative practice, the moments in which they were often led to satisfying and novel musical ideas. They were also frequently used in unintended ways, which allowed us as a group to incorporate a level of indeterminacy into our performances and gave us the impetus to pursue new and exciting improvisational avenues. The features that were not incorporated into our practice tended to rely on functional programming in Python and involved more complex syntax. While I personally felt that this would give the ensemble a greater level of flexibility and control in our performances, such as the `map` and `transform` methods for the player-key data structure, it did require a higher level of technical knowledge, which was not there for all members of the group. It was clear from the user evaluation that this disparity created an environment in which the performances did not always feel collaborative and increasing the complexity of the code was one of the fundamental barriers to achieving flow during performances. This starts to answer two of this thesis' research questions regarding language as a collaborative tool for live code. It seems almost trivial to say that simplifying a collaborative language's syntax would lower the barrier to entry to ensemble live coding but it seems that it also helps facilitate the collaboration itself. Simple code allows performers to better manage the various facets of performance, such as keeping track of co-performers'

actions and connecting musical elements through code, as well as staying in flow.

This chapter has seen the addition of several new functionalities to the FoxDot language, which arose from practice with the Troop software that allows users to share the same body of text as they work. The combination of FoxDot and Troop has enabled TYPE to collaborate at a deeper level and create dynamic musical relationships with our code. However, research has suggested that “individuals working separately generate many more, and more creative [...] ideas than do groups” (McGrath, 1984, p. 131) and perhaps working in one shared text buffer stifles creativity and the generation of new ideas. Can an interface be developed where live coders work individually but towards a shared goal, while still being facilitated by the linguistic features developed over the course of this chapter?

7. CodeBank: Public and Private

Working in Ensemble Live Coding

7.1 Introduction

This chapter introduces a collaborative interface for live coding entitled CodeBank that has been designed to facilitate interaction in ensemble performance while offering performers a “safety net” for experimentation in an attempt improve the overall quality of improvised music. Quality in experimental and improvised music, however, is often subjective. For example, failure, especially in the context of electronic and computer music, can have its own aesthetic properties (Cascone, 2000), which come in the form of glitches and crashes, but it may not be desired by the performer. CodeBank is designed to give performers more control over their performance and introduce only meaningful changes into the music.

CodeBank is an interface that offers performers a “private” workspace to experiment in before sharing their work with a “public” performance space. This allows performers to take larger risks with their code without the fear of disrupting the flow of a collaborative performance. Each connected user is synchronised to a single “performance server” that is dedicated to generating audio for an audience and changes made to code in a user’s private workspace can be heard through headphones. CodeBank implements a technique for managing collaborative projects in software development called version control, but in the microcosm of a live coding performance. Version control is the process of managing changes in data where developers contribute to a shared repository by “pushing” changes from their own private version and “pulling” the changes made by other contributors to keep up to date. This chapter posits the question, “how does private working in group live coding affect performance?”, which it aims to answer over the course of CodeBank’s development.

7.2 Motivation

Early sessions using the Troop editor were sometimes described as “chaotic” and even a “cacophony”, such as in Section 5.3.2. The confluence of several different threads of musical experimentation would often lead to a harsh juxtaposition in the music, which was rarely (although not never) the desired outcome. CodeBank is an endeavour to remove the elements of chaos and discord from real-time collaborative live coding while still facilitating musical interaction during

performance.

While researching collaborative digital musical interaction Fencott and Bryan-Kinns (2013) found that having their own digital “space” to work in, participants enjoyed themselves more when creating music together through a digital interface. The study required participants to create music together using the same interface but with three different control parameters; c_0 , c_1 , and c_2 . In the first, c_0 , all music modules were both audible and visible to everyone else in the session. The second, c_1 , was the opposite in that only data “pushed” to a publicly shared space could be heard or seen by the other participants. The last control parameter, c_2 , was the same as c_1 but participants could look at another user’s “personal space” if they explicitly chose to. Participants were then asked to fill out a questionnaire regarding their experience. Although not statistically significant, participants felt the best music was created when they had a private workspace ($c_0=5$, $c_1=12$, $c_2=8$) and they also enjoyed themselves more ($c_0=5$, $c_1=13$, $c_2=8$). Interestingly, participants felt they edited the music together the most when working with interfaces with private spaces ($c_0=4$, $c_1=11$, $c_2=8$) but also felt they worked more on their own ($c_0=3$, $c_1=10$, $c_2=13$). From these results Fencott and Bryan-Kinns suggest that splitting musical interaction into a public and private spaces should be a “key design consideration”, which has become the focal point of the CodeBank project.

The CodeBank editor also takes inspiration from popular version control tools, such as Git¹ and Mercurial², that are used in group software development projects. Version control keeps a history of changes made to a shared repository of code, including the identity of a contributor, and developers work on their own private version of the repository by “pushing” and “pulling” the changes made by themselves and their team. The parallel between private and public working in version control and the suggestions for musical interaction by Fencott and Bryan-Kinns is interesting considering that live coding shares traits with both software engineering and musical interaction.

CodeBank utilises private working and the testing of code before pushing it to the audience in contrast to the Troop interface in which all code was audience-facing and executed live. During a TedX talk³ Alex McLean stated that live coders are “not software engineers” as there is no problem to be solved and that they are “more interested in causing problems than solving them”. Alex is referring to the *process* of software engineering and what the code is used for but live coding uses the same basic tools, computer code, just in a very different way. Version control is usually utilised in projects that involve multiple files and large amount of data whereas live coders will often work within a single document. Thus the tools for collaborating in a software development project might not be appropriate for a music performance but why can’t they be appropriated to be used by ensembles in live coding? In the same video McLean describes a live coder as a “person

¹<https://git-scm.com/>, accessed 28/02/18

²<https://www.mercurial-scm.org/>, accessed 28/02/18

³<https://www.youtube.com/watch?v=nAGjTYa95HM>, accessed 28/02/18

on stage making code writing software on stage that generates music”. As writers of software it could be argued that a more collaborative live coding experience occurs when contributing to the same piece of software. However, without the use of a designated collaboration tool, such as Troop or Extramuros, live coders are only sharing sound, not code. That is not to say that live coders don’t experience musical interaction through sound, but that there exists another layer of communication that can be explored by bridging the gap between music and code.

7.3 Phase 1: Initial Implementation

7.3.1 Development

The design for the CodeBank interface takes its inspiration from the ‘History’ class (Rohrhuber et al., 2007) which is part of a SuperCollider extension called the Republic (de Campo, 2014). This system, most notably used by the live coding ensemble Powerbooks Unplugged, allows users to share sound and text across a network of laptops through the distribution of “codelets”. These are small bodies of code, typically one or two lines in length, that can be used, altered, and redistributed by performers to generate sound, stored on each machine in chronological order. Performers use shared resources but work independently from one another to collaborate. CodeBank works in a similar manner but the independent working is not public facing in the same way that it is with the Republic. In fact, CodeBank could be considered an inverted version of the Republic; the shared repository of codelets are the audience-facing interface for creating sound whereas the performers’ code is hidden from view.

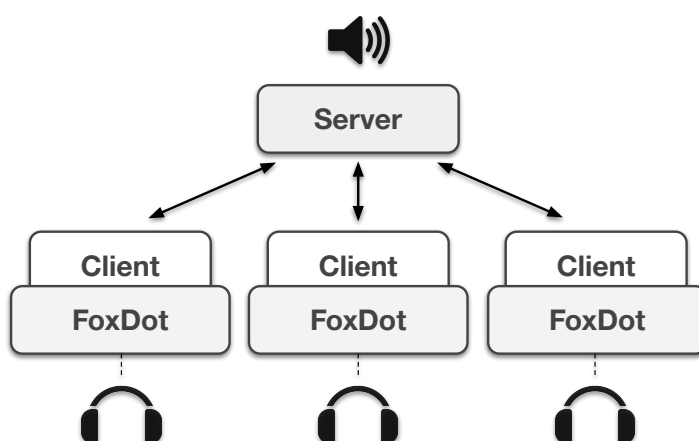


Figure 7.1: CodeBank’s Network diagram.

CodeBank, like Troop, uses a client-server network model. Unlike Troop, the server application plays an active role in performance. The server is connected to both the speakers and projector

and acts as the system interface to the audience. The CodeBank interface is built using Python's Tkinter library, which allows it to easily communicate with the FoxDot live coding environment. Figure 7.1 outlines the basic network topology for the system; client applications connect to the server, which in turn updates clients with new information received from other connected clients. Audio is generated on both the server and the client machines such that users can listen to the performance using headphones connected to their own computers. This also allows users to listen to changes made to code in their private workspace before committing it to the server. The client and server are synchronised using FoxDot's internal clock synchronisation methods⁴, which means performers can hear their private version of the code alongside the rest of the music in real time.

While CodeBank's mechanics for collaboration draw heavily on those found in version control, the implementation for pushing and pulling code to and from the server is slightly different. With version control software, such as Git, you are required to manually pull code from the shared repository to keep up to date whereas CodeBank automatically pushes new code from the server to all connected clients. CodeBank also implements a reservation system such that no codelet can be edited by different users at any one time to avoid conflicts in the code.

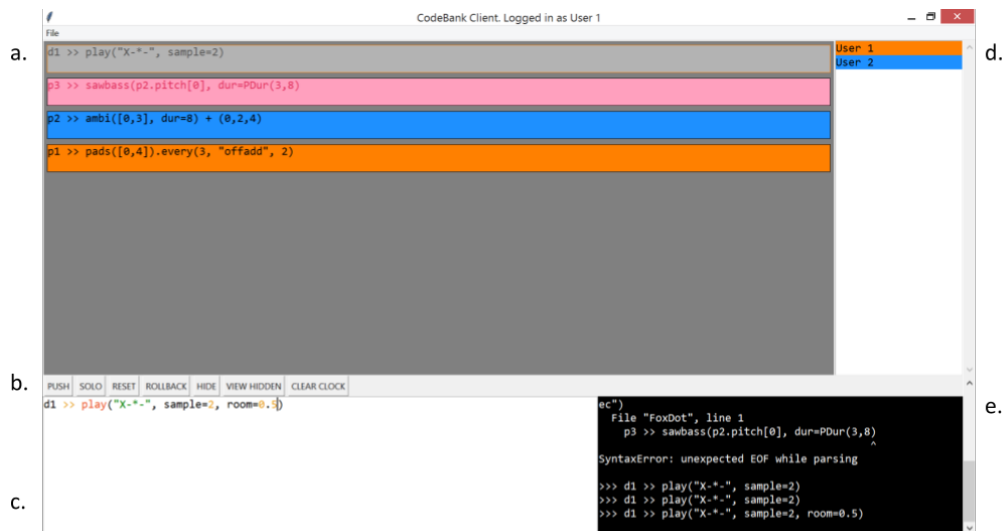


Figure 7.2: Screenshot of the CodeBank client interface.

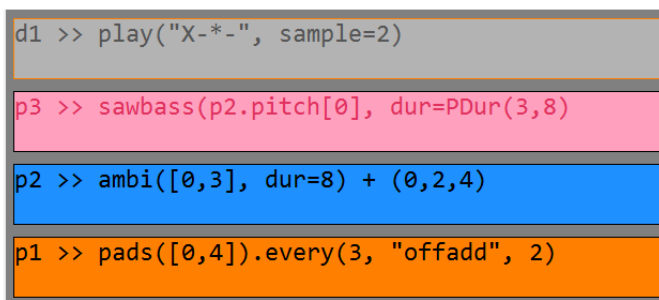
Client application

Figure 7.2 shows a typical screenshot for a connected client with five key elements labelled; the Code Repository (a), Action Buttons (d), Private Workspace (c), User List (d), Console (e). The User List contains information about the connected users and their associated colours. Each user's colour is used as the background for codelets that have most recently been edited by that user. In figure 7.2, User 1 is associated with orange and User 2 with blue, thus allowing each codelet's last

⁴<https://foxdot.org/docs/using-the-tempoclock/>, accessed: 29/07/2019

editor to be easily identified.

The Code Repository is the largest element of the user interface and displays information about the codelets being used to generate audio. This is considered to be the “public” facet within the public/private duality of the system. Figure 7.3 shows four codelets that have been added to the code repository, each with a different coloured background or text. When a user pushes a new codelet it appears in this window with its background set to the corresponding user’s colour. A users reserves a codelet for editing by clicking on it, which allows it to be edited without interference from other users. When a codelet is reserved its background and font are set to grey but its outline is changed to the respective colour of the user who has reserved it. For example, the top codelet in figure 7.3 is currently being edited by the user that is associated with orange, User 1. The second codelet has a pink background and font colour to indicate that this codelet contains a syntax error. In figure 7.3 the codelet is missing a closing bracket for the `sawbass` operation and changing the colour helps draw attention to this. Information about the error is also given in the console. All other codelets are displayed with the background colour associated with the user that last updated it.



```
d1 >> play("X-*-", sample=2)
p3 >> sawbass(p2.pitch[0], dur=PDur(3,8)
p2 >> ambi([0,3], dur=8) + (0,2,4)
p1 >> pads([0,4]).every(3, "offadd", 2)
```

Figure 7.3: Close up of the CodeBank Code Repository

The Private Workspace is a text box that allows users to interact with their code. In traditional live coding there is usually a keyboard shortcut, such as `Ctrl+Return`, that evaluates some, or all, of the code. This is the same for CodeBank but it only evaluates the code on the client’s machine so that the user can hear the effects of their changes within the context of the rest of the music. To interact with the codelets in the Code Repository, the user must use the Action Buttons, which define the possible actions that can be made by the user. The first button is the PUSH button that pushes the contents of the user’s private workspace to the server. If a user has clicked on a codelet, and consequently pulled it into their private workspace, then using the PUSH button will update the existing codelet instead of adding a new one to the repository.

The SOLO button can be used to help identify individual sounds or sequences. Fencott and Bryan-Kinns found that separating an interface into two music modules added a level of confusion and users found it difficult to isolate an element or sound. To counter this problem, the SOLO

mechanism was added to CodeBank; it uses the `solo()` function from FoxDot to group together any player objects in the private workspace and play only them, allowing the user to better pinpoint certain sounds.

The RESET button is only effective when a codelet has been pulled from the repository for editing. A user can press the RESET button to undo any changes and notify the server that the codelet is no longer reserved. Earlier iterations of a codelet can be accessed by using the ROLLBACK button. After a codelet is pulled from the repository, pressing the ROLLBACK button will display the code that was contained by the codelet before it was updated and pressing it multiple times will display previous versions. Like version control tools and The Republic's History class, this allows users to revert back and use older versions of code if they want to.

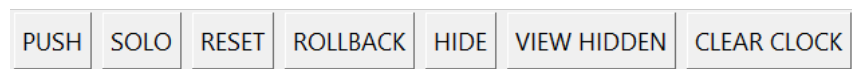


Figure 7.4: Action Buttons used for interacting with the CodeBank interface

If a codelet is no longer being used it can be hidden from view by pulling it from the repository and pressing the HIDE button. If a user wants to revisit a codelet that has been hidden, they can press the VIEW HIDDEN button, which will display all the hidden codelets. Pushing the codelet will remove the hidden status from the codelet, making it visible to all users again. To stop all sounds a user can press the CLEAR CLOCK button. Lastly, the console gives feedback to the user, such as error messages, but can also be used to display useful information through the evaluation of Python's `print()` command. Doing this in the private workspace will only display in the console of the local user and not clog up the console of other connected users.

Server application

The server-side application shares many similarities with the client but does not have a private workspace or console, as shown in figure 7.5. This is because it is the audience-facing component of the application and is designed to be displayed using a projector whereas the client-side application is not. Information about the code repository and user list is displayed to the audience so that they can see the colours associated with each user. This also allows them to identify which codelets are being edited (or have been edited) by which user.

One of the key requirements of the system is that the audio produced by both client and server applications is identical and, critically, temporally synchronised. The instance of FoxDot running on the server application is considered the “master” instance and the client applications must synchronise their clocks to it. This was achieved by extending the FoxDot `TempoClock` class and allowing instances to connect over a network and automatically update each other when one tempo

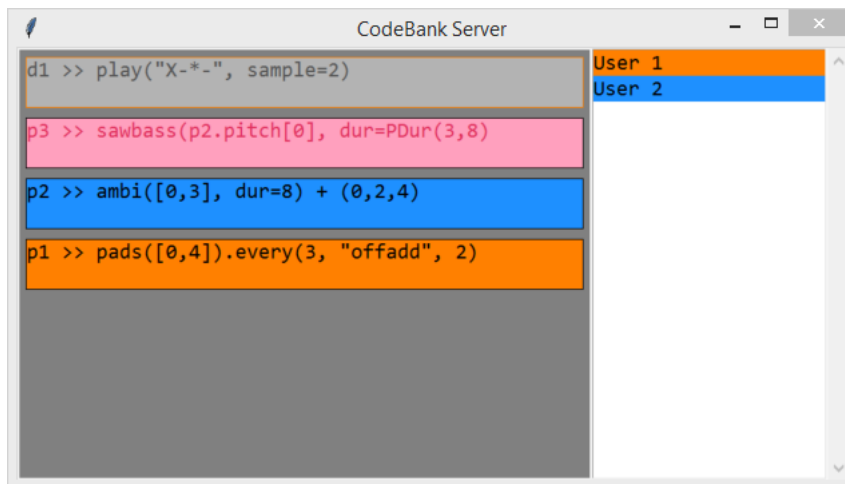


Figure 7.5: Screenshot of the CodeBank server interface

is changed.

One of the other challenges for replicating audio across client and server was ensuring that randomly generated sequences were identical on each machine. Live coders often utilise indeterminacy for creating melodic and rhythmic sequences, among other things, and if audio is to be identical across multiple users' machines, then the randomness needs to be replicable. Computer-generated random numbers are not truly random, but use algorithms to generate pseudo-random numbers. Python, for example, uses the Mersenne Twister algorithm which has a period of $2^{19937} - 1$ and can, essentially, generate an impossibly large series of values before repeating itself⁵. Making sure these pseudo-random series are replicated across multiple computers seems like a difficult task but the algorithm itself is actually deterministic given a number input, known as a “seed”. As well as using the server instance of FoxDot as a master clock, it can be used as a master seed to generate identical pseudo-random sequences on each client machine. Not only is this useful for creating indeterminate melodies and rhythms, but also for scheduling events with random periods. For example, a user might schedule a sequence to repeatedly reverse with unspecified intervals and these would need to be identical for each user to ensure audio is correctly replicated.

7.3.2 Practice

Rehearsal session, Sheffield - 09/12/18

Video recording: `ch7.1-CodeBank_Rehearsal-09.12.18.mpg`.

See Appendix A.8 for performance description.

For this rehearsal, the server was set up on a separate laptop connected to loudspeakers and users

⁵<https://docs.python.org/2/library/random.html>, accessed 24/10/2018

connected over a local wireless network. Headphones were used to listen to the audio output of code executed in private workspaces. The display with the server running on is not visible in the accompanying footage and, unfortunately, it is not possible to identify some of the sources of the sounds during this practice. As a result it was difficult to be specific about individual ensemble members' actions during the rehearsal.

This was the first rehearsal using CodeBank and we decided to not try and do anything differently to how we would normally rehearse in terms of process. However, listening back to the audio afterwards it seems we produced music in a very different way. It sounded succinct; there were not small iterative changes and it almost felt like it was an already edited and mixed track, not an improvisation. Comparing this to performances using Troop, such as the Together In Music conference in Section 6.3.2, there were no errors or mistakes that affected the musical quality (although the audio did cut out at one point) and it was kind of amazing to listen to this piece of music that was completely improvised but did not have the organic, free-improvisation aesthetic that all of our previous performances embodied.

7.3.3 Evaluation and outcomes

After using the CodeBank system, all performers noted that the style of live coding differed considerably compared to previous collaborative performances using Troop. Lucy stated:

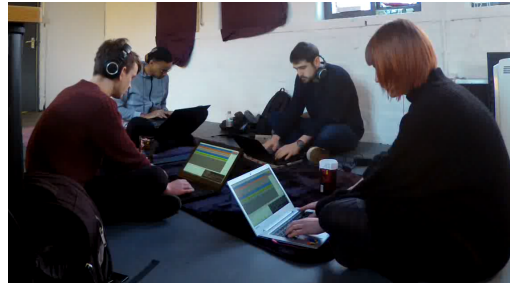
“It changed the way I was interacting with the code in that I was being more thoughtful about the changes I was making, but consequently paying less attention to what you guys were doing. Compared to using Troop where I have a general awareness of what you're both up to. I think it slowed me down a bit but also encouraged more significant changes rather than incremental ones.”

The slower process of coding meant that each time code was added or changed, it had more impact on the overall sound but that changes did not occur very frequently. There were several moments in the rehearsal where large amounts of time went by when every user was editing their own local version of code then pushing their changes to the public repository simultaneously, resulting in large shifts in the music. While this was an exciting process to be part of, it was also quite uncomfortable because the music would tend to change in a way you would not expect it to. Being able to experiment in your own workspace meant that any incremental change made to the code was only heard by the local user and would not give any indication to anyone else as to where the sound was headed.

The slower coding style also encouraged much more active listening compared to using Troop. Figure 7.6 shows two stills from the recording; one with members of TYPE wearing headphones to listen to the output from their private workspaces and another with the headphones removed to



(a) Using headphones to listen to the private workspace audio.



(b) Listening to the audio generated by codelets in the public repository.

Figure 7.6: Comparison of headphone use over the course of a CodeBank session.

better listen to the audio from the public repository. It was quite common for the group to remove their headphones and take a moment to listen carefully to the sound. This led to well thought out musical decisions that had more impact than the smaller incremental changes that occurred while using Troop.

It also emerged early on that there was a need for improving the user experience of the system, such as adding more control options from the keyboard. At this point, the action buttons could only be activated by clicking on them but live coders tend to use keyboard shortcut commands to control their interface and both Lucy and Laurie felt that using the mouse disrupted the flow of the session as they had to really think about using the mouse to control the interface. It was suggested that some of the more commonly used action buttons, such as `PUSH`, should be accessible from a keyboard shortcut to minimise the disruption. It would seem that the pre-existing tacit knowledge of navigating live coding interfaces using keyboard commands was so strong and ingrained in muscle memory that the cognition required to swap to a mouse or touch-pad would actually break the sense of flow.

7.4 Phase 2: User Experience

7.4.1 Development

After Phase 1, it was felt that the overall user experience of the CodeBank system could be improved. One of the priorities for users was to add better control mechanisms through keyboard shortcut commands. The first to be added was for pushing code to the server as this is one of the most frequent actions a performer uses. It was decided that the shortcut would use three keys, as opposed to two, so that it would be difficult to accidentally push code while editing it in the local workspace. The shortcut that was chosen was `Ctrl+Shift+Enter` as it is an augmented version of the shortcut for running code locally, `Ctrl+Enter`. Another action that users felt was disruptive was the act of pulling codelets from the public repository into the local workspace. This was

also accomplished using the mouse but both Lucy and Laurie felt it could be simplified through keyboard control. It was decided that using the **Alt** key in combination with directional keys could be used to highlight a codelet and pulled into the local workspace by pressing **Enter**.

Another addition to the program was a parser to disable users from changing metric attributes, such as tempo, in the local workspace. Doing so would change the tempo for all users as the tempo clocks were connected using the interface embedded in FoxDot and not in CodeBank. Using regular expressions (Goyvaerts, 2016), code could be stopped from being run in the local workspace that altered the `Clock.bpm` attribute, such as `Clock.bpm = 144` or `Clock.bpm += 10`.

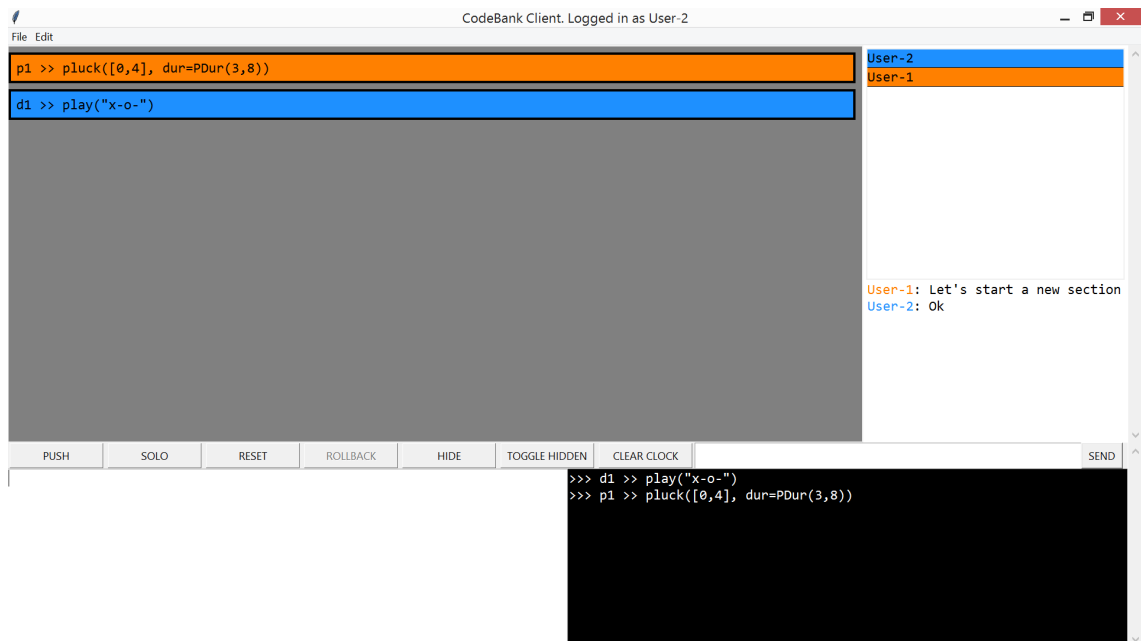


Figure 7.7: Screenshot of the updated CodeBank client application and chat box.

When using the Troop interface, code comments were often used as a way to discuss the music and what should happen next in a performance. However, trying to do this while using CodeBank was problematic. Users were much more focused on their own personal workspace when using the interface and, while they would appear in each user's console, comments would quickly disappear from view as more code was executed. This brought up the idea of adding an explicit "chat box" that would be reserved for discussion as opposed to code. CodeBank requires users to be co-located but communicating even short messages can be slow when done verbally. For example, one performer wants to share the message "let's start a new section". Signalling to other users to take off their headphones then telling them they think they should start a new musical section would be quite slow and cumbersome, but typing into a small box "let's start a new section" and pressing return would show this message immediately to all connected users who could either respond with a simple "ok" or looking at the first performer and nodding in agreement. Figure

7.7 shows the interface with chat feature implemented; there is a small text entry box to the right of the action buttons and a ‘send’ button next to it. Users pressing ‘send’ or the return key will post a message with their name into the box above. To make the discussion as transparent as possible, the messages are also displayed in the server application such that they can be seen by the audience. In other styles of improvised music making, such as jazz, the communication between performers is often quite visual and is part of the appeal to audiences so the decision was made to make all media for communication in CodeBank visible as well.

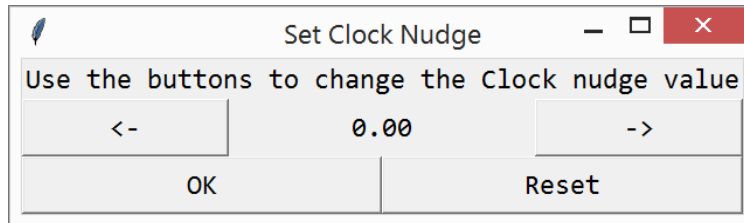


Figure 7.8: Window for adjusting the beat in FoxDot from CodeBank.

Another feature that was added to CodeBank as a result of user testing was a ‘clock nudge’ adjustment window. Some users found that the downbeat in their local workspace would occasionally drift from the public server by a few milliseconds. The tempo clock in FoxDot can be adjusted by small amounts if a user is out of sync with the server, but this is usually done manually with code evaluated explicitly in the private workspace. This was inconvenient and values had to be changed several times in order to synchronise clocks as tightly as possible. It also required that users memorise certain syntax to do this correctly. A specific ‘clock nudge’ window was added to simplify this process (see Figure 7.8) which allows users to increase or decrease the nudge value by 0.01 seconds by clicking on the the left and right arrows. Upon pressing the buttons, the new nudge value will be applied instantly and users will be able to hear the results and fine tune the level of synchronisation between client and server.

7.4.2 Practice

TOPLAP End of Cycle Party, Access Space, Sheffield - 19/12/18

Video recording: [ch7_2-TOPLAP_End_of_Cycle_Party-19.12.18.mpg](#).

See Appendix A.9 for performance description.

This was the first public performance using CodeBank and took place at Access Space; a charity-run venue that supports many artistic practices in Sheffield and has hosted several live coding events over the years.



Figure 7.9: Photo from the End of Cycle performance. Photo by Emily Stewart.

7.4.3 Evaluation and outcomes

This was a strange performance in many ways; we had decided to start by doing something “festive” as this was a Christmas party of sorts. This was more difficult than anticipated and it wasn’t long before we moved into the more comfortable territory of techno and EDM. The transition between these styles benefited from the slow and meaningful approach CodeBank enforces; it felt gradual and meaningful, which might have been difficult to achieve had we all been making lots of small, iterative changes.

As the performance reached the halfway mark we started to fall into old habits as the synchronisation mechanism stopped working and we relied on making more changes directly to the public code instead of working on the idea locally. This played a big part in changing the overall aesthetic of the music. Up until the 15 minute mark, listening to the performance felt more like listening to a pre-recorded industrial techno track rather than an improvised performance, similar to the experience from the rehearsal session in Section 7.3.2. Once we started to push codelets without listening to them in the private workspace we started to lose the cohesiveness of the musical quality, which is why the last 10 minutes sound so incongruous. We would often have periods like this using Troop where we would have to navigate the music to find a new idea to work on but the juxtaposition of this explorative style against the more structured and purposed music we had produced did not work well in the context of the overall performance. This was a shame as our take away thoughts from the performance focused on this and somewhat tarnished what was, overall, a very accomplished performance.

During the first 15 minutes of the performance every change to the code seemed meaningful and succinct. When the tempo was changed to 140 bpm and we moved onto a more percussive, sample based section (05:30), we discussed the change using the chat feature, which was a lot easier than trying to do it by talking. However, not all performers were aware of the messages appearing

in the chat window. Lucy felt that she “wasn’t really keeping an eye on the chat box” and added “maybe because of where it’s located? I was focusing more on the left of the screen where the code is”. She did go on to say, however, “I think the chat was good for audience engagement/interaction as we got a few laughs there”. At 18:10 Lucy and I have a discussion, the result of which is passed down the line to Innocent via Laurie through speech. Instead of typing in the chat box, we opted to communicate verbally, which took a large amount of time. It was clear that the chat box was not being utilised properly in its current state and needed to be improved.

From around the 10 minute mark in the video you can see three out of the four performers with their headphones around their necks and only occasionally putting them on for brief moments, as seen in Figure 7.10. It later emerged that, in the time leading up to this point, everyone experienced clock drift and were no longer in sync with the server. With complex rhythms at play it was hard to re-synchronise, even using the ‘clock nudge window’, and most of us only used headphones to listen to any larger textural changes made to the music in the private workspace.

One factor causing clock drift was the combination of changing tempo and network latency; the change in tempo always happens at the start of the next bar but the reference point may differ across laptops depending on when the code is executed. For example, let’s say that the current tempo is 120 bpm and the latency for sending data from the client to the server, and vice versa, is 0.25 seconds, i.e. 0.5 beats. A user pushes code to change the tempo to 150 bpm at 3.1 beats into a 4-beat bar. The code is sent as a codelet to the server where it is activated at beat 3.6 and schedules the tempo change for the start of the next bar, beat 4. The code is then transmitted to the other performers and activated at beat 4.1, which schedules the tempo change at the start of the next bar, which occurs at beat 8. By the time the client applications will have reached beat 8 (2 seconds later) and changed tempo, the server clock will already be at beat 9, one whole beat ahead of the client. To re-synchronise with the server, clients will have to set the clock nudge to 0.4 seconds (1 beat at 150pm), which, without calculating manually and understanding the problem that caused the drift, might take a lot of trial and error to solve. One way of addressing this issue could be to make the ‘clock nudge window’ made more sophisticated by allowing the user to pick if the nudge value is changed in seconds or in beats and use different increment values such as 1, 0.1, and 0.01. Ideally the process of adjusting for clock drift would be taken care of automatically and is something I am keen to improve for future performances.

One observation I made while watching the performance back was that the management of the codelets was done very well. Every so often one performer might take it upon themselves to clear unused codelets from the interface using the HIDE button, creating a clear canvas to work on. It was particularly useful having four group members as three could continue working on code if one was taking time hiding codelets. This meant that there was no slow down in terms of code creation while removing any unused code. We did not, however, make use of the ROLLBACK button that



Figure 7.10: Frame from video of Sheffield Algorave performance with CodeBank.

allows users to return to previous versions of a codelet’s history. We had not properly explored this in rehearsals and the practice of going back in time is not something we are used to doing as live coders. Very often our performances involve pursuing creative avenues as they appear and these often lead further and further away from the point of origin, but having the ability to backtrack could be an interesting approach in future performances, but needs to be explored further in group rehearsals.

The last five to ten minutes felt a bit more ‘lost’ than the earlier parts of the performance. We seemed to be stuck in quite a repetitive loop without a clear direction or end-point to aim for. I don’t know if this was due to a lack of coordination or inspiration, or if some of the technical issues with CodeBank’s timing mechanism played a part in this. My hypothesis is that taking our headphones off discouraged us from spending time working in the private workspace, which consequently led to less meaningful musical changes when compared to earlier in the performance. However, speaking to a member of the audience, they felt that it went very well; “It sounded like you knew what you were doing – everything gelled”. They also commented on how surprising it was that the level of coordination across multiple performers was so high; “I felt like it was more that, like, with one person, one person knows what they’re doing but with four people it would sound, sort of, uncoordinated. It’s cool how it didn’t sound like a mess of noise”.

7.5 Phase 3: Synchronisation and User Monitoring

7.5.1 Development

After our initial practice sessions in Phase 1, Lucy noted that, because she was putting more thought into her own code, she was paying less attention to the actions of her co-performers.

This happened on several occasions for everyone in both rehearsal and the performance in Phase 2. During performances with Troop the changes to the audio were, like changes to the code, incremental. In CodeBank, however, a user might change a codelet significantly within their local workspace and consequently create a large shift in the soundscape when pushed to the public repository. It is often difficult to navigate the improvisation during these more abrupt changes, which sometimes render the code in your private workspace obsolete. If good improvisation is like a good conversation, then this is like someone interrupting you mid-sentence with an off-topic question. To better manage this problem, a “user monitoring” function was added to the system that allows a user to listen to the incremental changes made by the monitored party from their private workspace. In the User List section of the client interface (labelled ‘d’ in Figure 7.2), boxes were added to the left of each user’s name that, when clicked, started monitoring the user, indicated by a black square in the box as shown in Figure 7.11.

When a user is monitored, all locally evaluated code is sent to the server and forwarded to the users who are monitoring them. The code is evaluated, rendering identical audio streams, and displays the code to the user in the console. Clicking the box a second time removes the black square and stops the monitoring process, such that the local user would only hear the the audio generated by their own code and the code in the public repository. Multiple users can be monitored simultaneously to give performers an indication of any upcoming musical changes.



Figure 7.11: User monitoring functionality of CodeBank.

As well as giving users the ability to monitor each other, the chat box was updated to grab more attention when a new message is received. The border of the chat box flashes with the colour of user who has sent a message, as shown in Figure 7.12. During the performance, information was mainly communicated aurally as opposed to through the chat box. This might be quick for passing on information to the person at your side, but passing a message down a line of several people can be quite slow compared sending a single message through the chat box. There were also some unintended side-effects caused by addition of the chat feature; the chat input box is located directly next to the CLEAR CLOCK action button that, when clicked, stops all sound. On several occasions in rehearsal, a user would attempt to click in the chat input box but accidentally press the CLEAR CLOCK button and silence everything. To stop this from happening, the CLEAR CLOCK action was moved to be only activated from the drop down menu, or executed explicitly through code.

During the performance several clients ended up out of sync with the server. This meant that


```
inno: an yeah, that was me
inno: thanks
Ryan: heh no worries
inno: I'm liking this
Ryan: im bopping away here
loz: \o/
inno: ooooooooooh
inno: funky
Ryan: nice
Ryan: Sorry to be a
downer, but I g2g soon :(
```

Figure 7.12: CodeBank chatbox with flashing borders.

performers spent large periods of time without wearing headphones and not utilising the private workspace. Consequently it felt as if the latter stages of the performance were not as well thought out as earlier sections. In an effort to address this, the timing mechanism of FoxDot was updated to better synchronise musical events across multiple computers. Prior to this, FoxDot would increment its clock's beat counter by 0.0001 beat every "tick" and activate a scheduled event (such as playing a note) when the beat counter was equal to the scheduled time. However, these "ticks" were sometimes inconsistent and two separate instances of FoxDot would drift out of sync over larger periods of time. This was changed to simply use the number of seconds elapsed since the last tempo change to calculate the current beat, which timings much more consistent across multiple instances of FoxDot. Furthermore, another issue discussed in Section 7.4.3 in which tempo changes were not coordinated correctly was also addressed with this update. Instead of just sending the beats-per-minute value over the network, the start time of the the last tempo change is also sent to connected instance of FoxDot so that, even if the tempo change happens at different times, the reference point for the change is the same.

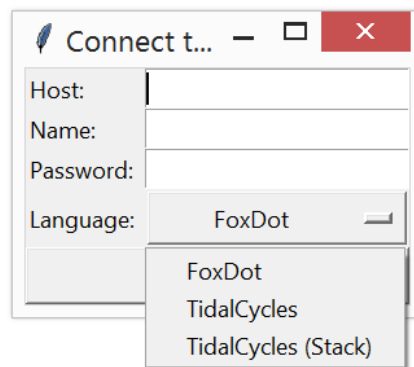


Figure 7.13: Login window for CodeBank with TidalCycles language option.

The CodeBank system has also been made language agnostic in the same vein as Extramuros and Troop, which allows a wider range of live coders to engage with this style of collaboration. One of the main goals of this research is to make collaborative live coding as accessible as possible but limiting these tools to work with only one language is contrary to this idea. Like Troop,

CodeBank’s functionality has been extended to interface with other live coding environments. However CodeBank’s action buttons are specifically tied to features in FoxDot, such as SOLO and RESET, which means these features need to be reproducible in the desired host language. To address this, ‘interpreter classes’ were added that contain the necessary code in the host language to be evaluated to reproduce the necessary functionality of CodeBank’s action buttons. So far the only classes that exist are for interfacing with FoxDot and TidalCycles as the actions such as SOLO and RESET are difficult to reproduce in other environments such as SuperCollider and Sonic Pi. When connecting to the CodeBank server, users are now presented with a drop-down menu (see Figure 7.13 in which they can select to live code with FoxDot or TidalCycles. If the language chosen does not match the one used on the server application, which is selected using a command-line flag, then the user is not able to connect to the server and is informed of the correct interpreter needed.

7.5.2 Practice

Late at the Library: Algorave, London - 05/04/19

Video recording: `ch7_3-British_Library_Algorave-05.04.19.mpg`.

See Appendix A.10 for performance description.

This was an Algorave event that took place at the British Library in London in association with the Alan Turing Institute. At the time of writing, it was the largest Algorave to ever take place in the UK, outside of a festival, with an audience of 700 people. TYPE were asked to perform at the event and we decided to use the opportunity to perform with CodeBank in its third phase of development. Unfortunately, Lucy was unavailable to play at the time, so only Laurie, Innocent, and myself performed at this event. I was hoping to use screen capture technology to record the feed from the CodeBank server laptop but due to issues with the display refresh rates at the venue, I could not mirror my displays and therefore could not record the screen. Furthermore, the video recording did not capture the screen contents correctly due to a high contrast between the dark room and bright screen, which means I am unable to accurately describe the code used during this performance (this is discussed further in Section 7.5.3).

7.5.3 Evaluation and outcomes

Speaking to some audiences members who had seen us perform before with Troop, they felt it was one of our best performances to date. Watching the recording back I definitely felt that the affordances of CodeBank lent themselves to creating a succinct musical performance. There were several distinct sections to music and the transitions between them mostly felt fluid. Much of the



Figure 7.14: Photo from Late at the Library. Photo by Coral Manton

performance was spent with a four-to-the-floor kick-drum rhythm underpinning bright chord stabs and melodies, although we were able to find periods of variety by moving to minor scales and experimenting with darker and more distorted textures.

An unfortunate aspect of this performance was the amount of reused musical material that featured from previous performances, including those we did with Troop. I was the main guilty party here and perhaps the pressure of playing to a larger audience with a new interface caused me to experience some performance anxiety and I fell back on musical elements that we had played before and knew would work. CodeBank was designed to encourage users to experiment with new material and create novel musical ideas by providing performers with a personal workspace to experiment in but in the end it is up to the performer to create the music.

This was a technically difficult performance for several reasons; we had to use some low-powered technology, which resulted with some synchronisation issues, and the volume of the PA system was, at times, too loud to hear anything through headphones. The first issue was brought on by the fact that CodeBank requires an additional laptop that is dedicated to running the server application. I used my personal laptop for this as it has a fast CPU and plenty of memory, which meant I had to borrow a laptop, which was much less powerful. The issue with computing power mainly concerns FoxDot and its synchronisation mechanism; adding CPU-intensive effects, such as a comb delay or reverb, delays the music by a fraction of second and the private workspace and public audio become de-synchronised momentarily until the local audio “catches up”. This is a huge problem for those who don’t have access to high-spec computers and ensemble live coding should not be reserved for those who can afford a spare laptop. Innocent also experienced issues with synchronisation early on in the performance, and found it difficult to listen to the output from his private workspace. This was particularly frustrating as it had worked perfectly when had tested the set-up earlier in the day. The second issue was caused by sound levels. It was difficult to hear the output

from CodeBank’s private workspace during the performance, and this was made more onerous by the synchronisation issues that arose. Laurie said the speakers in the venue “drowned out” the audio from his headphones and found he couldn’t tell the difference between the public and private audio once we had started working several different musical layers. This meant that much of the performance was spent without headphones off by all users, which is not a problem in and of itself, but when experimental and/or incremental changes are being pushed to the public repository, as opposed to being developed in the local workspace, it defeats the purpose of using CodeBank. Most live coding, including our own practice with Troop, is performed in this manner; making small changes that both the audience and the performer hear at the same time. However, CodeBank is designed to hide these incremental changes from the audience such that hear only a “final” version of a musical decision. When these two styles of live coding are combined it upsets the balance in the aesthetic of the whole performance. It could, of course, be argued that CodeBank itself upsets the aesthetic balance of conventional live coding by hiding the incremental changes away from the audience, but I will discuss this further in the conclusions of this chapter. A clash between incremental and more meaningful musical changes occurred during the performance at 28:00 when Innocent introduced a bell synth. He did so without using the private workspace and then began to make incremental changes to the codelet. At this point he could not hear output from the local workspace, so evaluating code locally would be pointless as he would only hear changes once pushed to the public repository anyway. Consequently, the incremental changes occurred slower than one might expect in a more conventional live coding performance because of the extra time needed to push and pull the codelet to and from the repository.

Viewing the video recording of the performance, one might notice that the screen projection is almost completely white. This is caused by the high contrast between the dark room and bright interface and meant that almost no on-screen action was captured. This a major problem for documenting CodeBank performances. The interface is also very bright compared to typical live coding interfaces and it doesn’t seem to match the aesthetic style of the late-night Algorave. If I were to develop another version of CodeBank I would add a “night mode” that sets the background of the interface to black with white text.

7.6 Conclusions

7.6.1 Personal reflections

CodeBank is designed to help reduce human error and improve the overall quality of collaborative live coding performance and, to some extent, this has been achieved. Before discussing this, there is also something to be said about these design goals in the context of the philosophy of live coding.

For many practitioners, live coding is “embracing error” and letting failure lead you in musical performance, but CodeBank arguably does the opposite. While it does provide a safety net for experimentation it also lets users try and fail with ideas without fear of doing so in front of a live audience. The CodeBank system actively encourages users to experiment and let error and failure guide them in performance but in a space they can be comfortable in doing so. This does lead to a delay between the formation of ideas and their eventual sonification for the audience and it could be argued that this reduces some of the “liveness” in live coding. The counter argument is that by allowing performers to experiment in a local workspace, CodeBank supports improvisation and the creation of spontaneous musical ideas. CodeBank also implements a far more complex interface than Troop and a user’s attention is split between several different key features, such as the local workspace, the public repository, and the chat box. The complex nature of the interface seems to place a larger cognitive load on the user and forces them to either spread their mental resources across multiple facets of the interface or sacrifice their attention on one feature to focus on another.

One of the research questions posed in this PhD is “how can collaborative interfaces be used to reveal creative processes at play in ensemble live coding performance?” and it has yet to be addressed in this chapter. I don’t feel that CodeBank reveals creative processes and, if anything, I believe it obfuscates them. From the audience perspective, codelets appear on the screen but the process in which they are developed is completely hidden. Even when one user edits an existing codelet, the only indication that a change has been made to the code is the change in background colour, indicating a new user has updated it. It is difficult to keep up with changes, especially as the codelet’s spatial location can change quite drastically when being updated. Furthermore, the codelets themselves only contain black text with no syntax highlighting; I would have liked to add this and make code clearer to both audiences and performers, but this would have taken a lot of extra work. Text in the codelet would have to be added one character at a time and placed in the codelet at specific positions in the x and y axes. Syntax highlighting would also have to be tailored specifically to each user colour to avoid code being illegible against various background shades. This would definitely have been possible, but I felt that cost of time and effort would outweigh the benefits.

Some features in CodeBank were not utilised very frequently, most notably the ROLLBACK button that, when used, would revert a codelet to its state before the last edit was made to it. The implementation of the feature was inspired by the collaborative software engineering tool, version control, and provided some interesting possibilities for performance, such as the re-introduction of previous musical themes. However, in practice, it was only ever used to revert simple changes to codelets, such as playing a stopped sequence. This was, in part, due to the fact that “rolling back” was not something that members of TYPE were used to doing in their own performance

practices, both together and apart. It is a novel idea but required a much more conscious effort to utilise and did not feel intuitive to use. It also required a level of learning, which is true for several aspects of the CodeBank interface, as the experience is very different to traditional live coding. CodeBank has many features that take time to know and understand, several of which are accessed via the action buttons. The use of buttons in live coding is very rare, as most interaction with the interface occurs through text and keyboard shortcuts, and this took some getting used to. As suggested by members of TYPE, several mouse-based actions, such as clicking on codelets to pull them into the local workspace, were also given keyboard shortcuts in an attempt to maximise the accessibility of the interface via the keyboard and tap into the ingrained tacit knowledge of keyboard-based control for live coding interfaces. However, not all of CodeBank’s functionality can be accessed using the keyboard, such as hiding and un-hiding codelets, and I think this is one of the main contributing factors to the larger cognitive load placed on the performer when using CodeBank.

There are also several technical challenges to playing with CodeBank; the first of which concerns the means of synchronisation between a performer’s local version of FoxDot and the version generating audio for the audience. Initial synchronisation of FoxDot instances can be quite straightforward and the user is given some level of control over it through the use of the clock nudge window, but problems arise as drift occurs over the course of a performance. This seems to be related to computing power. CPU intensive effects can result in delays to note onsets and cause them to fall out of sync with other instances of FoxDot. There is no method for rectifying this other than waiting for your laptop to catch up. This can be quite frustrating and, as happened in both our performances, can cause you to play without the use of headphones and push more incremental (and unknown) changes to the public repository. Furthermore, CodeBank requires any ensemble that uses it to also have an extra machine to host the server application. While computer technology is more affordable than ever, evident in such projects as the Raspberry Pi⁶ or Arduino⁷, these lightweight computers don’t often meet the specifications necessary to effectively run a live coding language such as FoxDot. This means that CodeBank requires access to an extra PC with sufficient computing power to be used effectively in live performance, which is not always possible.

I would argue that the CodeBank interface helped us create “better” music in the sense that musical contributions were polished and succinct, and errors and mistakes could be overcome in the local private workspace. However, the amount of communication while using CodeBank was less than when we used Troop and we combined to create novel musical ideas less frequently. For example, in the Algrave Assembly performance at the University of Leeds (discussed in Section 5.4.2), there was a very satisfying moment when “the combination of ideas” led to the creation of “a fresh musical sequence”. In CodeBank, however, it was closer to having complimentary musical

⁶<https://www.raspberrypi.org/>, accessed: 29/04/2019

⁷<https://www.arduino.cc/>, accessed: 29/04/2019

sequences being played together at the same time. This definitely helps create a seemingly well-crafted musical performance, but perhaps does not deliver the organic creative experience we are used to having while playing together with Troop.

7.6.2 User evaluation

Similar to the user evaluation of Troop, I asked members of TYPE to consider notions of trust, risk, flow, and immediacy (Gifford et al., 2017) and evaluate CodeBank as a tool for self expression and ensemble communication as well as compare these aspects of the interface with Troop. With Troop, there was a much stronger need to trust in the co-performers more than the interface itself so I started by asking if the feeling between co-performers differed when using CodeBank. Here is Lucy's response:

It's hard to know what people are up to, so... a little bit it's like... I guess it's trust but it's also like having that familiarity of playing together and being able to anticipate to an extent what kind of things somebody might be doing, erm, and that's something where I think particularly for us, like, as a group, because we've played so much in Troop and we have that, kind of, familiarity with one another, erm, and because with Troop, you can kind of see really easily how people, like, work with the code, I find that works quite well in CodeBank. I think if we'd used CodeBank first and then changed to Troop we'd probably have a really different dynamic as a group of players because we wouldn't... we wouldn't have that, kind of... I don't know how to phrase it, like... this kind of base level of just understanding would have developed differently. Like the understanding between us as players, would have developed really differently, because you wouldn't see the development process because with CodeBank you just kind of see the finished product

Without the experience of playing together with Troop and being able to learn each other's habits CodeBank may not have been as much of a success. The musical combinations in Troop came from reacting and adapting to incremental changes whereas in CodeBank we tend to only experience the "finished product" of a musical idea. Even when using the 'user monitoring' tool, it is difficult to know what is happening elsewhere in the ensemble, which is important in anticipating co-performers' changes. Lucy also suggested that CodeBank might not allow users to develop this level familiarity. Developing trust in your co-performers is extremely important and CodeBank may not be a good tool for facilitating this in an ensemble. This was felt by all of the ensemble:

Innocent: "When we perform as a group in CodeBank, it feels like you're in your own little space, erm, then you push out the code and... but the development process feels

very, erm, separate to the rest of the group. Erm, and, yeah, not knowing what someone else is doing makes it harder to anticipate, erm, what's gonna happen basically”.

Lucy: “I feel a bit like, in... in CodeBank it's more like... like, say you're in, like, a typical band and you have, like, a drummer and a guitar player and a bass player and a singer, it feels a bit more like that, like we're all in our own little world. So, like, I'm more likely to react to something you do by changing something I've done, whereas in Troop I feel much more likely to go and, like, tinker about with other people's code”

There is more of a sense of independence within the group when using CodeBank compared to using Troop. This impedes the inter-personal development of musical ideas and also makes it difficult to anticipate, and consequently react to, musical changes during performance. However, this does seem to promote a mindset of perfectionism when creating music with CodeBank, Lucy said “I'm much more of a perfectionist about my bit... And not, like , in a conscious way but that's... I think that is, reflecting how I use it I think ”. This does mean that the changes that are pushed to the public repository are more developed and audiences hear a polished version of everyone's musical ideas, instead of the small steps of the implementation of a musical idea, as it would occur when using Troop. However, only working with each other's fully developed ideas is not conducive of ensemble collaboration. Innocent said “being able to push out something that's more developed means that there's less room for other people to change that as well, which I think might be the reason why you go back and actually work on the stuff you created”. The notion of independence also seems to discourage users from making changes to each other's codelets as any change made might be felt as an attack on someone's “finished product”. On this topic, Lucy said “it also feels maybe like a bit more rude? 'Cause like, I mean it's something that someone's, like, crafted really perfectly and being like ‘actually I'm just gonna change that’ whereas everything's a bit more, like, in flux [in Troop] I guess”. Upon asking about how easy it is to achieve flow and feel like you've reached a groove in CodeBank in comparison with Troop, Lucy responded:

Yeah, I think with Troop it's just is easier because [...] everything is just a bit, like, speedier. But I think you're more likely to, like push something that's not right or that's a bit jarring, or like, break that flow in Troop, I'd say. Umm, but then it's also in a way, like, maybe a bit easier to fix stuff like that in Troop. ‘Cause you haven't, like, pushed a whole big thing that you've spent ages working on and someone else pushes their big thing at the same time and they clash, that would never happen in Troop. But small clashes happen more often

Evidence can be seen of these “small clashes” occurring in several Troop sessions discussed in the previous two chapters, such as the performance at the ICLI discussed in Section 5.5.2. As we

experimented with percussive samples during the performance we found that the “rhythms battled against each other” but we continued to explore the musical sequences that were emerging. On these smaller clashes, Laurie said “it increases the potential the music more drastically in certain unexpected directions, I guess”. Lucy also commented on this by saying that, in CodeBank, “if I push something that sounds crap I’ll just take it straight back and change it. Whereas I think with Troop we’re more, like, likely to kind of work with those – like Laurie says – like we’re more likely to kind of go ‘ok that element’s a bit weird but let’s pursue that and let’s see how we can, like, morph stuff from that’. I think we’re more like, kinda just more fluid when we’re playing in Troop”. Lucy also went on to say:

CodeBank feels a lot more like playing a solo gig to me and if you’re like talking about personal flow I think when I’m playing a solo set I feel I have a very different, like, feeling, which is that, like, there’s much more pressure and you don’t have that time to kind of step back and think about what you’re gonna do. Like you have to really be, like, on the ball and focused, which in a way, is a sense of flow, but it also means if something goes wrong you’re really taken out of it, whereas I think when we’re playing with Troop, and I would say CodeBank has a similar feel to that, but when we play with Troop, I don’t feel that, like, same intensity of focus, which, like, means you never get into that kind of real, like, depth of just, like, your heads in the code and that’s what you’re doing. But also it means you can’t get, like, snapped out of it either. It’s like a more, like, even state.

The achievement of a flow-state is important in good improvisation but the discussion above suggests it could be detrimental to collaborative live coding performance. Flow is often defined as “the state in which people are so involved in an activity that nothing else seems to matter” (Csikszentmihaly, 1991, p. 4), which can also include co-performers. Giving users a personal workspace allows them to momentarily forget about the audience and their co-performers and become engrossed in their own work, finding a temporary state of flow. However, it becomes an experience closer to playing a solo set than group jam. In contrast to this, members of TYPE felt that live coding together with Troop gave them a feeling of confidence in their co-performers which allowed them to take more risks, but perhaps did not enable performers to enter a flow-state. Instead, what we did experience may be closer to “peak jamming” (Swift, 2013), which is the rewarding feeling achieved by taking part in a good jam session that keeps us coming back for more. Our experience with Troop is also probably closer to “group flow” (Sawyer, 2006), which is a state of flow relating to a group of musicians that “can inspire musicians to play things that they would not have been able to play alone, or that they would not have thought of without the inspiration of the group” (Sawyer, 2015, p. 95). Research into flow often concerns the individual

but Sawyer believes that group flow is achieved as a collective unit, where “everything seems to come naturally; the performers are in interactional synchrony” and “each of the group members can even feel as if they are able to anticipate what their fellow performers will do before they do it”. The idea that we are achieving group flow within Troop also helps explain why Lucy doesn’t get “snapped out it” when things go wrong during a performance in the same way she does when performing on her own. Being in group flow allows us to be aware of our co-performers’ actions and have trust in the ensemble to navigate any errors, whereas encountering a problem while in an individual flow-state can be a huge disruption. Instead of chasing the feeling of a good jam session and achieving group flow, CodeBank encourages its users to work independently in the pursuit of a well-crafted final artefact. Innocent feels this is mainly caused by the expectations imposed on the performers through the CodeBank interface:

I don’t know, it’s just different to... to using Troop just because we interact and build up the music. Umm, I think with CodeBank there’s an expectation, because we know how it works – you can sample the sounds before it goes live, there’s an expectation of perfection, which means that, umm, the music doesn’t build up, umm, organically as it would with Troop when you might put out something weird, like Laurie said, and then work with that. In CodeBank, umm, the expectation, at least internally, would be, uhh, sounding polished because we’ve had time to play around with it first.

By giving users a private workspace to test out their musical ideas, CodeBank imposes an expectation of perfection in each musical contribution. This makes it harder to collaborate on codelets as they are rarely works in progress, but have a sense of finality about them. Members of TYPE were also asked about the immediacy of the interface, and to consider how easy it is to communicate with the rest of the ensemble. There were several comments regarding the use, or lack thereof, of the chat box feature that was added to the interface:

Lucy: “I feel like the chat box is a really really good idea in theory. In practice, I just don’t look at it, and again that’s partly ‘cause I’m like – what I talked about about being so focused on what I’m doing – like, I look in the chat box and there’s, like, loads of messages and I’m just like ‘Ohh I haven’t read any of those’. And I would say, like, in Troop [...] we just, like, type little notes, like, in the code, I’m just much more aware of what everyone’s doing in Troop including if they’re, like, typing little notes.”

Innocent: “Yeah I think there’s a lot more interaction, erm, in Troop, umm, and I’m the same – I don’t look at the chat box in, uhh, in CodeBank, erm, because I’m focusing on- on the code so much. So I look up and I see stuff and I’m like ‘oh, well, I missed all that’.”

Lucy: “So I would say, like, probably in CodeBank I’m more listening for changes, like, you know in Troop we might be like ‘oh let’s do a percussive bit’ and then do a percussive bit. I think in CodeBank I’m more listening for changes so I’m more like ‘oh, what are they up to?’ erm, so I guess more like a traditional way of, like, improvising with other musicians; by ear.”

The independent flow-state achieved by members of the ensemble causes individuals to focus solely on their own work and become incognisant of the rest of the interface, including the chat box. This makes ensemble communication very difficult in CodeBank. With Troop we are able to add comments next to specific parts of the code to communicate and draw attention to certain aspects of the performance whereas in CodeBank it’s not as easy to relate a specific chat message to a specific part of code. Of course, other methods of musical improvisation, such as jazz, do not have such explicit modes of communication and, perhaps, should not be relied so heavily upon. However, with each performer so heavily invested in their own work, other non-verbal modes of communication are also difficult to achieve without the chat box and performers are reacting more to what they hear than what they see. Innocent found that communication was so slow via the chat box, he found himself using verbal communication more during the performance at the British Library:

When I first started playing with Troop, erm, it was just pretty different, like, seeing messages, umm, in there and, like, just little notes and little jokes and- but that was a fun element to it. Erm, and with CodeBank I’ve found myself, erm well in, like, the British Library Algorave, I found myself talking to Laurie, erm, a bit more rather than, like, putting in a little note, umm, in there. Umm, but yeah, I think the communication aspect is- is really fun, umm, and in Troop, erm, it’s a lot better and a lot more immediate, erm, than CodeBank.

He also stated that the communication using Troop was not only better and more immediate, but also more enjoyable than in CodeBank. Communication in CodeBank is very functional, whereas the lighthearted (and often off-topic) discussion in Troop also fostered a positive social atmosphere within the group. This helped develop a sense of ‘team spirit’, which aids in achieving group flow. Lucy also states that she is much more aware of the rest of the ensemble’s activities, including explicit chat messages, when using Troop as opposed to CodeBank, and this is another likely indicator that Troop is better at facilitating group flow than CodeBank. It seems that being cognisance of one’s co-performers’ actions is a pre-requisite for developing group flow in live coding, but this would not always be possible while in a hyper-focused individual state of flow. That being said, group flow and individual flow do not have to occur simultaneously, as Sawyer states, “[t]he group can be in flow even when the members are not; or the group might not be in flow even when

the members are” (2006, p. 159).

7.6.3 Final thoughts

Some interesting comparisons can be drawn between the themes that have emerged from the reflections discussed above and the results from the study conducted by Fencott and Bryan-Kinns (2013). In the study, most users felt that the overall quality of the music improved when given a private or personal space to work in and a similar feeling is shared by TYPE with respect to the musical output created in CodeBank. The ability to spend extra time perfecting a sequence helps improve the aesthetic quality of the music. Fencott and Bryan-Kinns’s study also found that users felt they worked more on their own when given a private space to work in, which was also true for TYPE when using CodeBank. Members of TYPE experienced flow on an individual level much more in CodeBank than they did with Troop, which often resulted in a much more independent creative style. However, the study also showed that users enjoyed themselves more and felt like they were editing the music together when given a private workspace, which was not necessarily in line with how CodeBank users felt. The private space seemed to create a disconnect between performers in CodeBank and placed an emphasis on contributing more fleshed-out musical ideas instead of developing them together. Furthermore, the less formal channels of communication in Troop improved the social aspect of playing music together, whereas in CodeBank performers were so focused on their private workspace, explicit communication existed only in a functional capacity. This is probably one of the main reasons that members of TYPE enjoyed using Troop more than CodeBank, but was never considered as part of its design. Simplifying the interface and making performers’ contributions public-facing should facilitate both social interaction and, consequently, the achievement of group flow and should be integral to the design of future collaborative live coding interfaces.

The first research question posed in this PhD is “how can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?” and the development of CodeBank has provided an interesting answer. I was interested in improving how collaboration was facilitated but, based on answers given by members of TYPE, inter-personal collaboration is actually impeded when using CodeBank. Despite this, the overall quality of music generally improved. Performers tended to introduce musical sequences as “finished products” and this made it difficult to collaborate while developing these ideas as there was very little room for change and a larger sense of ownership of individual contributions. Furthermore, the emphasis placed on independent working in CodeBank, which did help ensemble members achieve a sense of flow and consequently benefited individual creativity, decreased performer’s awareness of their co-performer’s actions. This included messages sent via the chat box section of the interface, which is an explicit channel of communication. Even though much of our ensemble communication was

non-verbal and implicit, such reacting to musical changes, the chat box exists to communicate ideas quickly and directly. Not being aware of messages in this part of the interface means performers might not always be informed with respect to larger musical decisions, such as starting a new section. Moreover, the chat box is also the only facet of the interface that allows for informal communication; one of the most enjoyable parts of our rehearsals and performances when using Troop. In Troop, the “chat” is present within the code itself and it became quite a performative action whereas, in CodeBank, the code and chat is separated and less focus and attention was placed on this mode of communication.

By developing CodeBank and reflecting on practice in this manner, one thing has been made clear; the interface used *does* make a difference in terms of collaboration. Separating the coding activity into private and public spaces shifted performers’ priorities, fostered more individual than group flow, and even affected the quality of the music created. If asked to choose between Troop and CodeBank to collaborate with, the answer would depend on what the ensemble’s performance goal is, and I think that it is important that an ensemble gets to have this choice. The goal may be oriented towards the playing experience (achieving group flow and peak jamming) or the creation of high-quality music and the ensemble can choose the interface better suited to achieving success. Members of TYPE felt that Troop allowed for better and a more immediate sense of communication and was also more fun to use. While CodeBank has its advantages, it did not give the group the same level of satisfaction and enjoyment as when coding together within the same text buffer. Troop is also a far more accessible interface and requires no extra technology to run. But can the Troop interface be improved upon? In an attempt to make ensemble live coding accessible to as many live coders as possible, both Troop and CodeBank have been adapted to work with multiple live coding languages, but these languages cannot be utilised within the interfaces at the same time. To bring a broader range of live coders together, could an interface be developed that allowed multiple languages to run together simultaneously?

8. Polyglot: A Multilingual Interface for Collaborative Live Coding

8.1 Introduction

This chapter introduces a cross-language live coding interface, Polyglot, that allows multiple users to easily collaborate with one another without requiring a shared knowledge of the same live coding language. Currently live coders who wish to use interfaces for collaborative live coding, such as Troop or CodeBank, are required to know the same language, which means for every available live coding environment there are less potential collaborators to work with. Polyglot is a direct response to this problem and aims to give live coders with different skill-sets the opportunity to work together using the same medium that made Troop so effective. Polyglot uses much of the same technology that was developed as part of the Troop interface in Chapter 5 but extends the software to allow concurrent collaboration across multiple text buffers, and live coding languages, simultaneously.

8.2 Motivation

As has been discussed throughout this document, research has shown that music is an inherently social activity and making music together with others has been known to be beneficial to one's mental well-being (Clift et al., 2010; MacDonald, 2013). The focus on improvisation in live coding often leads to it being compared to jazz improvisation (Aaron et al., 2011) and if good improvisation is often referred to as a “musical conversation” (Monson, 2009, p.76), how can you have a good conversation if you aren't speaking the same language?

One of the main obstacles to ensemble live coding with Troop and CodeBank, among other collaborative interfaces, is that all members of the ensemble must use the same language. There are many different languages available and more are being developed every year. This reduces the chances of knowing the same language as another live coder, making it difficult to find suitable collaborators. This is not to say this cross-language live coding does not exist. In fact, members of the pioneering Algorave act, Slub, have always worked together using their own individual environments, only sharing some metric information over a network to perform together (McLean, 2015). However, from my own experience, and the discussion in Section 2.2.3, this practice does not reflect the majority of live coding ensembles, which tend to work homogeneously with regards

to language.

While many skills in programming are transferable across languages, there are also different programming paradigms which require the user to think differently about approaching the problem they wish to solve. The constraints of the problem itself are often the main consideration when choosing a programming paradigm or specific language and we have already seen that there are several live coding languages based in either functional or object-oriented styles of programming (see discussion in Section 6.1). Creative musical practice is not an exact science and there is no “one size fits all” method for approaching it. The co-existence of these two programmings paradigms in live coding is a testament to this and indicates that there will likely always be multiple languages to live code with. In a more traditionally musical sense, there are also multiple ways to represent musical notation; while the Western hemisphere may be used to the note and staff system, there are a variety of musical and rhythmic representations from around the world including Indian tabla drumming notation (Courtney, 1994) and the Japanese Kunkunshi notation style (Thompson, 2008) among others.

There are often technical issues when attempting to “play together” in a live coding context, especially when musicians are using different programming languages. Most types of music require performers to be in time with another, which, when live coding, often demands a level of technical knowledge to set up if there is not a mechanism for doing so built into the software. For example, FoxDot allows users to connect to a master clock, which measures the latency between machines and sets and adjusts the tempo accordingly as long as the users’ machine clocks are already synchronised using an existing protocol. However, FoxDot has no knowledge of how other live coding environments, such as TidalCycles, handle timing and would not be able to synchronise with it unless done manually by ear. Protocols for synchronising computer clocks have existed for some time, such as NTP, but the flexible nature of tempo and metre add a level of complexity on top of the basic clock synchronisation problem. Ogborn (2012) addressed this issue by developing a the EspGrid software, which is an implementation of a peer-to-peer tempo synchronisation protocol and does not require users’ machine clocks to be synchronised using NTP or any other means. Each performer runs an instance of the EspGrid software on their machine, which works behind the scenes to calibrate the relationship between the local machine clock time and the beat counter run by EspGrid. Users can query their local EspGrid instance for information about the tempo and current beat position in the metric grid to synchronise different music software, namely live coding environments, together over a network. In his paper, Ogborn, discusses the idea of “helper objects” that are built into the various pieces of music software to make it easier to communicate with EspGrid, which already exist in popular software such as ChuckK, Max, and SuperCollider and can be easily developed for other technologies. In Polyglot, multiple live coding languages and their respective EspGrid helper objects are combined into a single application that enables

collaboration while keeping all live coders tightly synchronised.

8.3 Phase 1: Initial Implementation

8.3.1 Development

Polyglot is the aggregation of several existing technologies for that were built for live coding; collaborative text editing like that found in Troop, temporal synchronisation through EspGrid, and audio generation from various live coding environments (typically using SuperCollider in some capacity). Figure 8.1 is a network diagram for a standard Polyglot set-up that outlines how the various facets of the system are connected. At its core, it is a collaborative text editor application similar to Troop that uses a client-server model to allow users to send keystroke and mouse-click information over a network. Unlike Troop, Polyglot allows users to edit multiple text buffers at the same time within one window. Each buffer corresponds to a different live coding environment that Polyglot sends code to in order to generate audio. An instance of the EspGrid application runs in the background on the client machines and calculates metric information using its own synchronisation algorithm. Each live coding language uses this information to set up its own time-keeping mechanisms and generate music in time with the others.

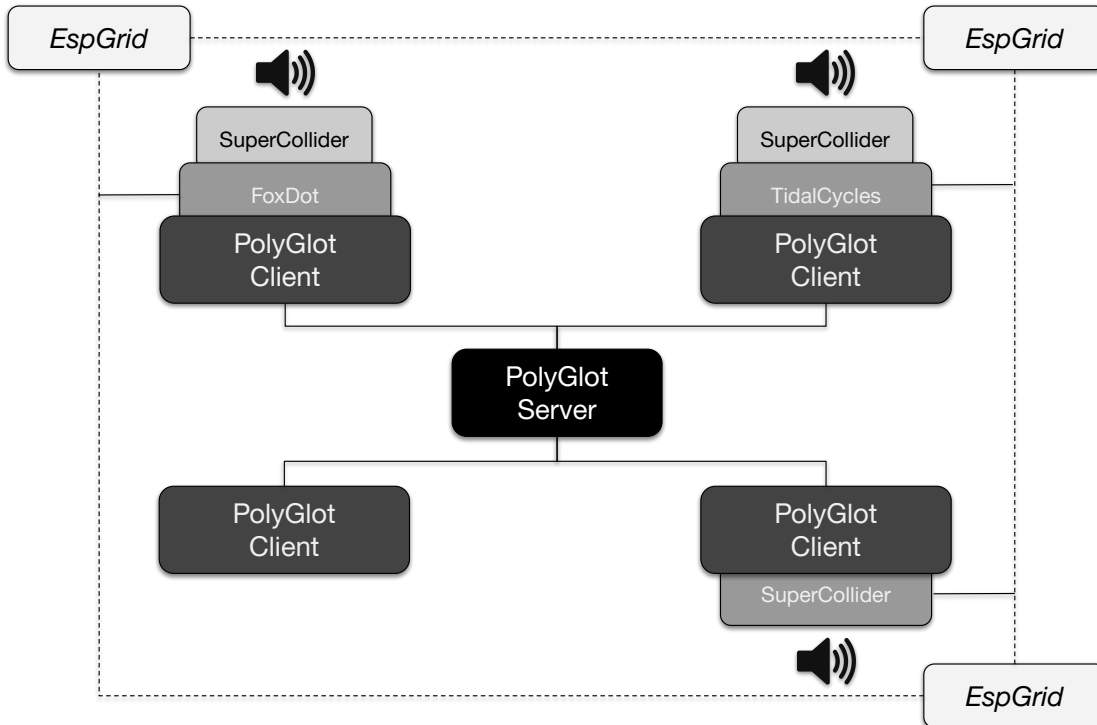


Figure 8.1: Polyglot network diagram.

The Polyglot interface can communicate with three live coding environments; FoxDot, TidalCycles, and SuperCollider. SuperCollider is made up of a live coding language, sclang, and an audio

engine, scsynth, and the latter is utilised by both FoxDot and TidalCycles to generate sound. Therefore SuperCollider is required to be running on any machine that generates audio but this complicates proceedings when running multiple live code languages on the same machine. Audio is generated in SuperCollider when it receives OSC messages with unique ID numbers from FoxDot and TidalCycles but any message received with a duplicate ID number is discarded. Furthermore, running multiple interpreters simultaneously consumes a large amount of computing power and can overload the SuperCollider audio server, and is usually best avoided.

The set-up in Figure 8.1 shows each live coding interpreter being run on a different client machine and one client not running any interpreter at all. Polyglot gives users the flexibility to only run the interpreters they need to on their own machine. Distributing the audio across multiple computers in this way enables the maximum amount of computing power to be allocated to each language and alleviates the issue of duplicate message ID numbers. Upon starting the application, users are greeted with a login window (shown in Figure 8.2) that lets them select the “active languages” they wish to run on their local machine. Users participating in the performance without running an active languages can interact with a “dummy interpreter” that will send code to their co-performers without executing it locally.

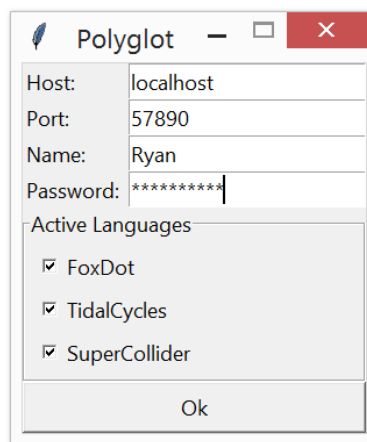


Figure 8.2: Login window for the Polyglot interface.

As an application that has much of the same functionality as Troop, it makes sense to re-use the same code-base instead of writing the Polyglot software from scratch. Polyglot extends the Troop software by using multiple text buffers, which also requires the server application to host multiple instances of the operational transformation server-side program (see Section 5.4.1 for a discussion on how this works).

The client application uses one text buffer for each interpreter, FoxDot, TidalCycles, and SuperCollider, which are also accompanied by a console for displaying information and error messages

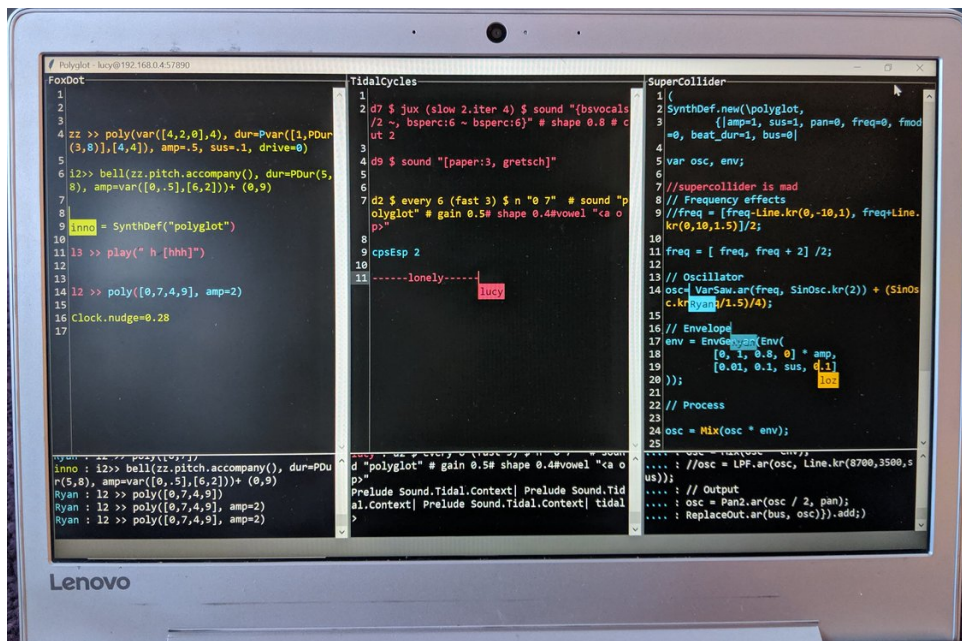


Figure 8.3: Photo of Polyglot being used with 4 users. Photo by Lucy Cheesman.

being returned by each language. Synchronisation of the audio is handled by the EspGrid software, which runs in the background as a daemon program and has to be started manually by the user before starting Polyglot. EspGrid automatically finds other computers on the network running the program and then agrees on a metric grid, which is accessible through its API. Each language requires a mechanism for accessing this information that Ogborn calls “helper objects” and are “necessary and not hard to create”. Ogborn himself has developed a helper object for the SuperCollider environment, and has also helped TidalCycles developer, Alex McLean, create one for the language. As part of this project, a helper object for the FoxDot environment has also been developed that regularly requests the tempo and metric grid from EspGrid and updates FoxDot’s scheduling clock accordingly if it has changed. This proved to be a difficult task as EspGrid did not work as expected on the Windows operating system, which is used by all members of TYPE. However, Ogborn was very willing to listen to feedback and improve the program accordingly over several weeks of correspondence, updating, and testing. The EspGrid software does work best if all users share the same version of operating system but perhaps this slight pitfall can be improved in time and allow any live coder, regardless of their set-up, to collaborate.

8.3.2 Practice

Rehearsal session, Sheffield - 07/05/19

Video recording: ch8.1-Rehearsal-07.05.19.mov.

See Appendix A.11 for performance description.

The first recorded use of Polyglot came at a rehearsal prior to an upcoming performance at the Festival of Algorithmic and Mechanical Movement (discussed in Section 8.4.2). Due to prior commitments, only Laurie and myself were present at the rehearsal but we managed to set up and synchronise Polyglot correctly. Both Laurie and I are most comfortable using the FoxDot environment but have some experience with TidalCycles and SuperCollider’s audio synthesis programming. We did not, however, have proficient skills in SuperCollider’s pattern library, which is used for scheduling note events, and decided to approach the rehearsal slightly differently to how Polyglot was intended to be used. Instead of separate audio streams connected to each text buffer, we decided to use only two, FoxDot and TidalCycles, and to use the SuperCollider text buffer to create a SynthDef that we could trigger from the other environments. This allowed us to live code in a way that we had never done before. It should be noted that the recording for this rehearsal is incomplete as the camera ran out of battery, but both Laurie and I were so engrossed in what we were doing that we failed to realise this for over thirty minutes.

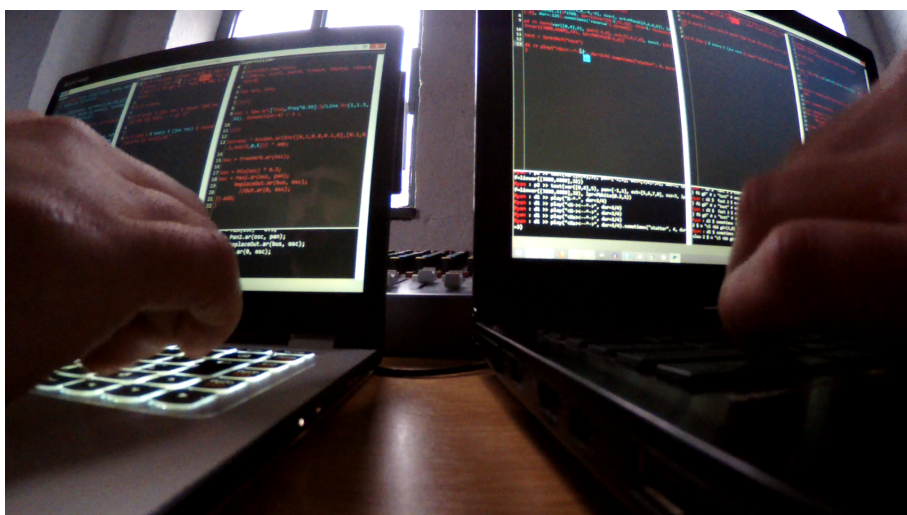


Figure 8.4: Screenshot from a Polyglot rehearsal recording.

8.3.3 Evaluation and outcomes

The music we produced in this session was rooted very much in minimalism, although this was not explicitly our intention. By using SuperCollider as an interface to control timbral parameters of the music we shifted away from our usual creative process of writing algorithmic notation for multiple pre-written synths to exploring the use of a single texture embedded in repetitive rhythmic and melodic structures. The constraint of only using one SynthDef forced us to think much more about the sound as opposed to form and prompted us to produce music that warrants a deeper level of listening in order to follow the slow transformation of the synth’s qualities.

Both Laurie and I agreed that this was a completely new way of live coding for us and was

only really made possible because of the Polyglot interface. Being able to interact with the music at both the structural level (through FoxDot and TidalCycles) and the timbral level (through SuperCollider) at the same time was extremely engaging and doing so collaboratively made it even more entertaining. As mentioned above, we were so engrossed in this style of music creation that we did not realise the power had run out of the camera for over thirty minutes. It was new and exciting and powerful. Neither Laurie nor I consider ourselves to be sound artists and would not rate our audio synthesis skills very highly, but this was a very fun and engaging way to experiment with synthesis and learn how changes at the oscillator level effect the overall music. This was a completely unintended outcome from the Polyglot project but has definitely opened the avenue to novel and fruitful collaborations between different styles of live coding.

There were still some issues with the interface that could be improved, however. With each performer hosting a different live coding language, it was not possible to get feedback in the console for the language *not* being hosted on the local machine. If I was to print out the list of available SynthDefs in FoxDot, I would have had to look at Laurie’s screen to see the console output. Similarly, if Laurie wrote TidalCycles code that contained syntax errors, he wouldn’t know this until I saw it in the console on my screen and told him. This would be an even greater problem to those connected to the server and not running any live coding languages on their machine as they would receive no feedback in the console whatsoever. It would be a great addition to give users the ability to run live coding environments locally without synchronising with the EspGrid software so that they could receive better information from the respective live coding language.

8.4 Phase 2: Language-Specific Feedback

8.4.1 Development

One of Donald Norman’s rules for user-centred design is “use technology to make visible what would otherwise be invisible, thus improving feedback and the ability to keep control.” (Norman, 1998, p. 192). In the rehearsal session in Phase 1, the feedback was limited to only the locally hosted language and not the others. This makes it difficult to identify syntax errors or see key values such as tempo or scale. To combat this problem, a feature was added to Polyglot that allows users to select languages they wish to be “active” on their computer but not generating audience-facing sound output.

This is achieved through the login interface, shown in Figure 8.5, with the use of tick-boxes prior to the main interface being opened. Languages with the the “sync” option left un-ticked will not run any code required to connect to the EspGrid. This is not a perfect solution, however, as the communication between Polyglot and SuperCollider takes place over OSC with no means of

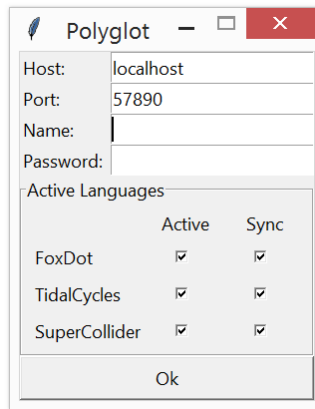


Figure 8.5: Updated login interface for Polyglot

retrieving the program’s response. Therefore it is impossible to provide users with feedback from SuperCollider from within the Polyglot window. In spite of these issues, it is still a step in the right direction for the design of the interface.

8.4.2 Practice

AlgoMech Festival, DINA Club, Sheffield - 18/05/19

Video recording: `ch8_2-AlgoMech.Festival-18_05_19.avi`.

See Appendix A.12 for performance description.

The Festival of Algorithmic and Mechanical Movement, i.e. AlgoMech, takes place each year in Sheffield and usually hosts an Algorave evening. TYPE were invited to perform and we decided we would use the Polyglot interface in our performance. Unfortunately, only one rehearsal with Polyglot involving all the members of TYPE took place before this event and not everyone felt comfortable using multiple languages during a single performance, especially with regards to editing SuperCollider SynthDef code. Instead of combining the multiple languages together into a unified performance style, as done in the rehearsal with Laurie and I discussed in Phase 1, we decided to approach the performance from a more pragmatic viewpoint.

The motivation for Polyglot’s development stems from the desire to remove the barriers to collaboration that are created as a result of the many different live coding languages that are available. Although it does provide a novel way to interact with, and combine, multiple live coding languages into a single performance, it exists primarily to allow two or more users to use different live coding languages from within the same interface and it should be used as such as part of its user testing “in the wild”. For this reason we decided to use the languages we felt comfortable using for the majority of the performance and then try a more experimental approach towards the end by incorporating the SuperCollider language and editing the SynthDef source code.

8.4.3 Evaluation and outcomes

In many senses this was a difficult performance; there were issues with the projector and laptops freezing and this made it difficult to find the level of group flow that we usually achieve when performing together. We had some trouble keeping audio synchronised at a tempo other than 80bpm (EspGrid’s default tempo) which was frustrating as our Algorave performances tend to fall under the umbrella of techno music and we would often set the tempo to around 128bpm. To pick up the energy level, though, we found we could play everything at double speed so the music felt like it was being played at 160bpm, producing fevered drum and bass sections. It was great that we were able to adapt to this constraint at times but it did push us out of our comfort zones for much of the performance unfortunately. We were also not able to make the most of the SuperCollider interpreter and spend time developing textures in the way Laurie and I did previously, which was a real shame but this was an Algorave performance at the end of the day and we were still able to combine TidalCycles and FoxDot well to produce a set of improvised algorithmic dance music.

It is difficult not to focus on the technical difficulties that were encountered during the performance, most notably was Innocent’s unresponsive editor. He was unable to enter characters as the interface froze on more than one occasion. This issue was highlighted by the fact that his was the laptop connected to the projector, which displayed all of these problems to the audience. This is likely caused by the inefficient implementation of the operational transformation algorithm used to display updated text. In Polyglot and Troop, the entire contents of the text buffer is deleted and replaced whenever it is updated. Consequently, the entire contents needs to be coloured every time a character is added or deleted and this takes longer as the total number of characters in the buffer increases. This process seemed to be manageable when only using a single shared text buffer in the Troop interface but the performance suffered greatly when three text boxes were being updated simultaneously. This was likely the cause of both the freezes and the location of the the text cursors to update incorrectly (as shown in Figure 8.6). This brings to mind the problems that occurred with Xerox’s pioneering development of the mouse and GUI desktop; “the benefits [...] were completely outweighed by the slow response speed. The display could not always keep up with the typing.” (Norman, 1998, p. 181). Just as it was the case for Xerox, the core idea behind Polyglot was not necessarily bad but the engineering solution used did not do it justice. It does highlight, however, that computational performance should be taken into account when designing more complex software for making music. There are two possible ways of addressing the problem of Polyglot’s poor performance: one is to completely re-implement the source code using a programming language faster than Python and the other is to improve the current algorithm being used. Given the amount of work it would take to achieve only a possible benefit, the latter appears to be the best choice of action. Two potential ways of improving the Polyglot algorithm

are as follows:

- Update the contents of the text boxes not in use by the local user at a much slower rate, reducing the frequency of deleting and replacing its contents.
- Instead of replacing the contents after each update, update the text box at the location of each character insert/delete.

The feedback system during this performance was not perfect either, again for two reasons. The first is that users have to install all of the software for which they want to receive feedback. If two users who each have a different language installed on their laptop wish to use Polyglot to play together, they would have to install the other language in order to get information printed to their consoles. A remedy to this would be to send the feedback information to any users that are not currently running that language. This way, a user who does not have FoxDot installed could run `print(Clock.bpm)` to view the tempo from within the FoxDot text box and see the results on their laptop. There is still an issue with this proposed feature, as there is still no way of receiving feedback from the SuperCollider interpreter over OSC. However, a command-line version of SuperCollider's interpreter, `sclang`, does exist and can be activated as a Python subprocess. This opens the possibility of combining the two communication channels into a single process that is accessible to Polyglot. The `sclang` process could be started from Polyglot, which would listen for code input over OSC, and output any feedback into the Polyglot console.

Another unfortunate aspect of the performance was linked to the synchronisation of the two sound-producing laptops. In the sound check several hours prior to the performance we were able to connect to EspGrid successfully and change tempo without the two computers going out of sync. However, this was not the case when it came to the live performance and we were stuck on 80 bpm for the entirety of the performance. This was frustrating as we were performing at a dance-music event and tempo is important when setting a beat. It was still possible to play everything twice as fast to achieve the feeling of a 160bpm tempo, but this was faster than what we would have liked to use. One alternative to crossing our fingers and hoping EspGrid will behave would be to use just one laptop for audio output as the EspGrid will synchronise all local languages very effectively as there is only one machine clock being considered. The downsides to this would be the lack of feedback to other users not hosting the languages (although remedied by the additions discussed above) and the computational strain put on one computer. The latter may lead to high CPU usage and crashes in SuperCollider but it would solve many issues with synchronisation and is an avenue worth considering.

Given the technical difficulties, the performance went well but the multiple disruptions made it difficult to find a sense of flow and there were several moments where the music stagnated for minutes at a time while these technical issues were being addressed. For example, the opening

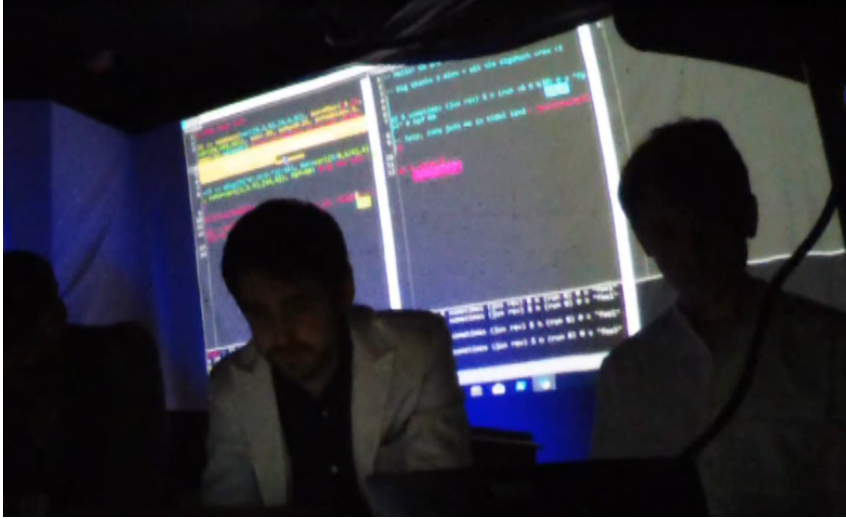


Figure 8.6: Screenshot of the AlgoMech performance with the blurring text cursors.

few minutes of the performance consisted of a simple drum beat and repeated baseline groove. Although repetitive, the combination of percussive sounds from TidalCycles and synth textures in FoxDot complimented each other well. As the performance went on we were able to embellish the music in the higher registers and managed to hit a good jam centered around the synth baseline. As the pitch of the baseline started to change more frequently, the default major scale didn't match the darker, grittier tones we were creating with and we moved into a minor scale. This transition took place mainly using FoxDot; the ability to adapt to something we didn't like proved to be quite difficult when not using an unfamiliar language. Once we had made the transition, we were able to add more percussive elements using TidalCycles with more confidence. With the added confidence came more succinct and varied musical elements. The fevered drum beats created in TidalCycle combined well with the dark synth riff's being generated using FoxDot to produce some great moments of industrial drum 'n' bass. This was a performance that centered around confidence; perhaps working with more simple pitch patterns gave us more confidence to work with textures more freely and also the ability to do so using a language we were less familiar with. It is hard to feel confident when you are standing in front of an audience with your computer freezing, and that likely also contributed to the more 'stagnant' sections of music that occurred alongside the technical disruptions.

As we attempted to use the SuperCollider editor later in the performance, we were unable to debug errors and struggled to make the most of it's ability to work with the oscillators in more fine-tuned detail. We did, however, manage to experiment with the sound textures using FoxDot but it would have been interesting to hear the effect of a live manipulation of a SuperCollider SynthDef, given the integration with both TidalCycles and FoxDot. Even with the number technical difficulties, though, I really enjoyed using TidalCycles in a performance. I don't have the confidence in my abilities with the language to perform solo with it but using it alongside the ensemble

and within the context of Polyglot where I can also use FoxDot made it easier and gave me the confidence to try new things. I knew Lucy would be able to help with syntax errors and that the group would be able continue to make music if I didn't know what to do. I think with time and practice the confidence in my abilities with TidalCycles and SuperCollider would improve through using Polyglot as a platform for sharing knowledge and group learning.

8.5 Conclusions

8.5.1 Personal reflection

Polyglot is a much more technically complex piece of software compared to Troop and this was, ultimately, its undoing. In theory, Polyglot allows live coders to collaborate regardless of whether they use the same language but incorporating three text buffers into one editor perhaps proved overambitious in practice when using laptops with limited resources. There were times, particularly when one or more text buffer contained a large amount of text, when keyboard inputs would take upwards of 5 seconds to be processed. During the performance at Algomech Festival, Innocent even found that characters would take 30 seconds to a minute to appear on his screen. Donald Norman notes that for many pioneering technologies this sort of problem is not unusual; the potential power of the system is not matched by the technology available and so does not deliver results with adequate performance.

The ambitious nature of the project led to the creation of an exciting and novel tool but, for Polyglot, “the spirit was willing but the implementation was weak” (Norman, 1998, p. 181). Troop was created using Python to better integrate with FoxDot and much of its code was re-used in developing Polyglot. Python is not an efficient language and as applications written in it become more complex, the performance decrease becomes more noticeable. The engineering solution used needs to be informed by the complexity of the system and, given what I know now about the complexity of the Polyglot system, I would have opted to write the program in a language optimised for speed, such as C++.

Polyglot was originally designed as a means for multiple live coders to collaborate together in the highly effective way that was facilitated by Troop, even if they did not know the same live coding language. However, after the success of the experimental rehearsal session that took place with myself and Laurie (described in Section 8.3.2), we felt that Polyglot also afforded us the opportunity to showcase a level of multilingual virtuosity in a performance. Unfortunately, each ensemble member had varying levels of confidence with each language and, coupled with the technical issues described above and in the previous section, this resulted in a very disjointed performance. The alternative approach to the performance would have been to return to Polyglot's original purpose

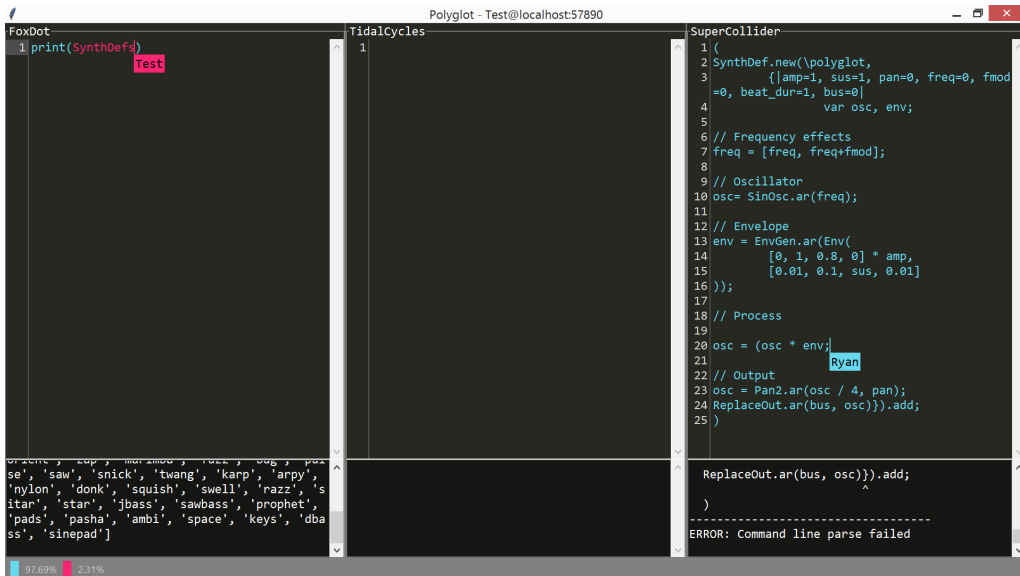


Figure 8.7: Polyglot interface showing feedback on an error in the SuperCollider tab.

and to collaborate using only the language that each performer was most comfortable with, but that would have resulted in Lucy using TidalCycles by herself and the rest of the group collaborating within FoxDot. Without utilising SuperCollider as a manipulator of sound synthesis algorithms, then any performance, given the context of this ensemble (a group of live coders that regularly play together only using FoxDot), would have felt closer to a technical demonstration than a musical improvisation. The true potential of Polyglot will only be exhibited if all three languages can be combined together in a structured and well-thought out piece or if it used as a facilitator of collaboration for live coders of different languages. The AlgoMech performance in Section 8.4.2 was the final performance scheduled as part of this research project and, unfortunately, organising another that involved other live coders using SuperCollider and TidalCycles was not feasible within the available time constraints of this PhD. This is partly to blame on my own shortsightedness as I prioritised collecting observational data of the same ensemble using a new interface over utilising the software in the context for which it was designed.

With regards to the research questions posited in this thesis, Polyglot provides some interesting potential answers. Addressing the first question, “how can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?”, it seems that cross-language collaboration is possible, and can be very fruitful, when using Polyglot. However, just as it was the case in Chapter 6, there was a disparity in the level of technical knowledge between ensemble members that, when combined with a resource-intensive and unresponsive program, created a difficult performance environment. There is definitely potential for Polyglot to be used as a powerful tool for cross-language collaboration, particularly when utilising SuperCollider’s relationship with FoxDot and TidalCycles as their sound synthesis engine. However, for Polyglot’s potential to be fully realised, it will have to be run on more powerful machines in its

current state or re-written in a more efficient language.

Much of Polyglot’s functionality is identical to Troop, such as the coloured fonts that help audiences identify contributions, but is spread over three separate text buffers. This is similar to the set-up in the Extramuros interface and so suffers from the same issues; the creative processes at play are separated and difficult to see from the audiences perspective. While it may seem that separating the coding activity into three separate text boxes may obfuscate the ensemble’s interactions, the coloured fonts in Polyglot are consistent across the text boxes and audiences and performers can still see who has written what and where.

8.5.2 User evaluation

As with previous user evaluation sections, the concepts of “trust”, “risk”, “flow”, and “immediacy” were discussed in an interview with members of TYPE regarding the Polyglot interface. When asked about the level of trust in Polyglot to take risks during performance, Lucy made an interesting point about how her risk-taking was not only constrained by the interface, but also about the performance setting itself:

That was a high pressure gig situation, like, we were on late and sandwiched between two really like high energy, like, dance floor acts that obviously makes you feel like you have to bring that energy as well and we were all, like, tired and a bit overexcited, like me personally I can barely even remember it [...] So I was just like ‘play it safe’ and we’d had issues in practice and stuff not sounding good between Tidal and FoxDot so I was just like ‘play it safe’ in that gig 100% and that was partly the software but also partly the context.

She went on to elaborate that she felt that we were not as well prepared for the performance as we should have been and we probably would not have performed using Polyglot if it there wasn’t the need to collect performance documentation for this research. She did go on to say that “considering the stress levels of that gig, it actually sounded pretty good, like all things considered. There was a lot of pain going on”. Part of that “pain” was regarding the slow performance of Polyglot, which dramatically affected the feeling of immediacy for the performers:

Innocent: “Well for me, umm, I had issues with, sort of, lags, umm, it would start out OK but it gets to a point where it’s so slow and everything is taking about 30 seconds to show up on the screen after I type it that it- it’s just not working at all, so that’s the only issue that I had but you guys didn’t seem to have that problem.”

Lucy: “I didn’t have it as bad as you but it was sometimes, like, 20 seconds, like, everything would just stop and you’d have to wait for it to refresh so that’s obviously challenging”

Have to wait 20-30 seconds to be able to express your ideas in the code is frustrating and does not make for a successful interface for musical improvisation. This lack of immediacy in the interface also disrupted the ability to achieve flow during the performance. Lucy stated:

I think, like, that the fact that it was so slow to respond made that [achieving flow] quite challenging. I would say also, I had a lot – because I was sort of thinking about the Tidal window more – I lost a lot of the awareness of what you guys were going, which, like, is just a bit jarring for me sometimes when you’re just like ‘oh where’s that sound come from?’ but then I realise Innocent’s written a whole new Player and I hadn’t even noticed he was typing it, like, which happens sometimes when we use Troop but I think it was more distinct. For me that is an important part of the flow; that awareness. [...] It’s a completely different way of working because you’ve got different languages and in terms of, like, the interactivity and the flow and the collaborative elements it’s harder. [...] Switching between languages is, like, just makes your brain short circuit a little bit sometimes

Not only did Polyglot’s slow response speed affect the group flow, but the increased cognitive load of having to think about multiple languages and multiple text buffers made it difficult to achieve flow during the performance. Laurie made a good point about the arrangement of the text boxes, that we were “just so used to looking just vertically at the lines within that window”. Perhaps an alternative interface layout would have improved the ability to keep track of co-performers’ actions and maintain a level of flow in the performance. The difficulty of using multiple languages simultaneously was noted by the group, but this did not deter them from seeing the creative possibilities that could be available from further practice with the interface:

Lucy: “I think inherently it’s got huge amounts of potential to be an even more exciting tool than Troop but I just think it- it needs that time investment from us and we haven’t been able to give it that yet.”

Laurie: “Bit more of a learning curve, definitely, because you need three times the expertise”

Lucy: “Yeah and a completely different way of thinking about the music that you make”

Innocent, in particular, found having multiple languages present in the same interface difficult to cope with and actually felt that the presence of code he did not fully understand was detrimental to the practice.

Innocent: “Umm, there isn’t any benefit to, umm, to that tab [TidalCycles] because I don’t understand it, umm, but I think not being able to understand what’s going on is

also a bit of a barrier. Because if I'm- if we're all using FoxDot and, umm, and we've got our players and got our percussion I can look at what's going on and understand, umm, where the sound is coming from, umm, whereas if the percussion is coming from Tidal then I don't really understand what's going on or where the sound is coming from, umm, I don't know it just feels, I don't know..."

Ryan: "Is it exacerbated by seeing code that doesn't make sense to you? Is it more confusing than if you were just listening and couldn't see the code?"

Innocent: "Yeah I suppose. If there was this percussion bit going on and I wanted to create another percussion bit, if it was in the same language I could see how the other one is built and then I can build mine to be in line with that. Umm whereas when I don't know how it's being created in Tidal then I can't. [...] It makes it difficult to create a new line of code to sit nicely alongside what's already there"

Lucy: "It's an added layer of complexity and I would say that even knowing both FoxDot and Tidal, that was hard I think there was also like, I think the sounds are really different in Tidal, like it's got a really different sonic character. That means it's really hard to get things like to work nicely together."

This "barrier" created by the use of incomprehensible code from other languages draws similarities with the user evaluation in Chapter 6, in which members of TYPE felt that some FoxDot syntax was overly complex and its use in performance only by some members did not foster a sense of true collaboration. This could also be the case here where only some members of the group are able to effectively code in the other available languages in Polyglot, resulting in some ensemble members feeling ostracised and an un-collaborative environment. Furthermore, the each language has a different "sonic character", which makes it even more difficult to create complimentary music across the different text buffers. Innocent did suggest that more uniformity, such as using the same set of samples, may have helped and a similar sentiment is shared by the others regarding visual information during performance:

Innocent: "If you were able to use FoxDot samples in Tidal then, I guess, I would also be able to see at a glance... a bit of a recognition and I could see where that sound is coming from"

Ryan: "So the visual aspect is important for collaboration?"

Lucy: "I would say definitely, for me I'm, like, not a great musician. If we were sat there playing instruments I would find it much harder to be like 'ohh it's in this key' [...] Just being able to look and be, like, 'right, Ryan's using these numbers, I'm gonna use those numbers and it's gonna be the same'. It's much easier for me."

While the visual element of collaborative interfaces is beneficial, it is also important that all members of the ensemble are able to understand its contents to ensure the most fruitful collaborations and help cultivate a feeling of group flow. My intuitions were that the rehearsal session with Laurie was more successful than the live performance at Algomech Festival, and this may have, in part, been due to the similar levels of technical knowledge that we had in both the TidalCycles and SuperCollider languages. I asked Laurie what his thoughts on that session were:

Laurie: “It was interesting just being able to tinker with the core generator of the sound. It’s something that I really really enjoy – I don’t understand it nearly enough to do that freely and have that translation process where I can think of the sound and modification and put it straight down.”

Lucy: “Yeah because there’s another layer because you need to understand, like, sound synthesis first principles basically don’t you?”

Laurie: “Yeah and then work out how that’s represented in the SuperCollider code, which is another thing, erm, but I really enjoyed using the same SynthDef in both FoxDot and Tidal because, again, that kind of – that has that unification we’ve been talking about. It actually did happen, and what I was saying about changing one thing and it trickle down to all the different, erm, players that was incredible”

Lucy: “Yeah that’s an area where I think there’s huge amount of potential with Polyglot”

Laurie also felt a similar level of satisfaction in that session and even described the experience as “incredible”. Using the same SynthDef in both FoxDot and Tidal also helped address the problem with clashing sonic characters of the two languages and Lucy recognised this a potential strength for the Polyglot interface for the future.

8.5.3 Final thoughts

One of the most important points to make is that Polyglot was never truly used for its intended purpose over the course of this chapter. It was designed as a collaborative interface to allow live coders to collaborate at deep level regardless of the language they wished to use but was utilised in practice as a demonstration of multilingual live coding virtuosity. Unfortunately, TYPE are not yet virtuosos in more than one live coding language and the combination of multiple languages in this style of performance context was actually detrimental to performers’ experience, disrupting both the feeling of group flow and collaboration. Ideally Polyglot would have been also have been used in practice in a context in which three users each used a separate language to play together, synchronised using EspGrid, but, due to time constraints, this was not possible. However, it is clear

that there is potential for Polyglot to also become a tool for cross-language collaboration at a deeper level than just temporal synchronisation and shared code; it can also unify the languages' "sonic characters" by sharing SuperCollider SynthDefs in both FoxDot and TidalCycles and editing them live in the SuperCollider text buffer. This technique was briefly explored in a rehearsal session and the process of combining multiple languages was described as "incredible" by Laurie, suggesting an exciting possible avenue of creative interaction that could be pursued in future. However, Polyglot did suffer from poor performance; in certain situations characters could take up to 30 seconds to appear on screen after being typed and users' cursors would appear to lag and blur. This made performing with the interface difficult and slow and there was little sense of immediacy. Perhaps another implementation of the software built with a faster programming language than Python would be able to handle the functionality better but it is unlikely to be used effectively in a performance setting in its current state, although it does provide the foundations for a promising method for cross-language live coding collaboration.

9. General discussion and conclusions

9.1 Introduction

The main aim of this research was to produce software that explored aspects of both ensemble performance and computer programming and examine how they are learned, used, and adapted through practice-led research. To achieve this, two research questions were posited based on the rationale outlined in Section 3.2, and are listed below:

1. How can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?
2. How are collaborative interfaces used to reveal the creative processes at play in ensemble live coding performance?

To answer these questions, four interfaces, including a language, for collaborative live coding were developed and iteratively refined over the course of this project. Each interface was an exploration into an aspect of ensemble interaction within the context of live coding, which has the unique characteristic of using text as part of its performance medium. Unlike most other forms of music-making, this text can be used to explicitly communicate information between performers and the audience. The line between verbal and nonverbal musical communication is often blurred; code is often self-descriptive, which allows the musical intention to be directly interpreted provided you understand the basic syntax. The way technology mediates this communication plays a crucial role in how a live coding performance is both delivered and received, and this notion forms the basis on which this PhD research has been conducted. This chapter presents a short overview of the timeline in which this research was conducted before discussing some final reflections on the work and addressing the main research questions followed by my closing thoughts.

9.2 Timeline

Figure 9.1 is a Gantt chart outlining the development of each interface starting in March 2017 and ending in June 2019. Each asterisk also marks the location of a recorded rehearsal or performance that has been included in this thesis. The chart illustrates the iterative approach used to develop the interfaces, with feedback and reflection being used to inform the subsequent design. While it may seem that there were roughly the same number of performances with each interface there were closer to 15 performances, and many more rehearsals, that took place using Troop during this

time. Troop and FoxDot were developed together over a 27 month period whereas the duration of the development and practice for CodeBank and Polyglot were just 9 months and 3 months respectively. Although it appears that a disproportionate amount of time was spent working on Troop, it should be stated that large sections of the time associated with its development were spent writing code that were reused in the subsequent interfaces. For example, TYPE had no public performances between April and November 2017 while the operational transformation algorithm was being implemented, which was re-used as part of the Polyglot software. During these periods, however, TYPE did continue to rehearse together and started to develop chemistry as an ensemble through using Troop. As will be discussed further below, this could be a contributing factor to the group’s reluctance to use subsequent interfaces as we developed an affinity for the style of performing that we had developed, which became ingrained in the Troop software as time went on.

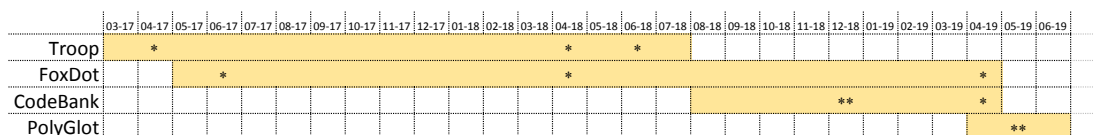


Figure 9.1: Project gantt chart. Each asterisk marks the occurrence of a documented performance or rehearsal.

9.3 Discussion

When I started this journey I approached it from mainly an engineering perspective; I asked myself “how do I design the best user interface for collaborative live coding?” and I set about developing software that I thought would make the process as simple as possible. I thought that if it was made easy then of course the collaboration would be easy also. While there exists a large body of literature on HCI for computer supported co-operative work, the unique blend of musical performance and computer programming makes live coding difficult to contextualise in this field. The variety of network music systems discussed in Section 2.2 is a testament to the breadth of interfaces used for collaborating in computer music. Music – even electronic music – is about people at its core and musical collaboration is about people’s ideas working together. In the end this journey has been primarily about gaining tacit knowledge about how to improvise with other people while trying to create a digital platform that helps facilitate that process. I have come to learn that there is no one-size-fits-all “secret recipe” to creating an interface for collaborative live coding and that there is no substitute for practice and chemistry when creating music.

This is not to say that the software I designed had no effect on the music. Just as an instrument will affect how music sounds, the medium we use to make music together will affect how it *feels*

to create it. If you are in a state of flow and the software doesn't behave as you would expect it to – perhaps it is slow to respond or enters characters in the wrong order – then that flow state can be disrupted. These are issues with performance and implementation as opposed to flaws in the software's design, but in a domain that is embedded in technology as much as live coding is they do need to be considered when looking at the final product. In Section 8.5.2, for example it is clear just how much Polyglot's poor responsiveness hindered achieving flow in performance from both Innocent and Lucy's comments. However, the impact of the interfaces on performance was not always negative. In a performance style that requires performers' eyes to be locked on their screens, bespoke collaborative software does help with the sharing of information and ideas. Without the ability to see co-performer's actions it is difficult to share and build on each other's ideas; a key part of group improvisation. Being able to do this in real-time through the Troop and Polyglot software enabled performers to inform their own creative decisions in the same way an improvising jazz musician might infer creative intent through physical gestures (Seddon, 2005).

While the development of the collaborative interfaces has been a key focus of this research, they are not works of art in themselves; it is the musical performances created using them that are, in fact, the creative artefacts. Through these performances I have developed new tacit knowledge in *how* to collaborate as a group of live coding improvisers, which has not only provided me with better a understanding of my own practice but also guided the design of subsequent collaborative interfaces. But, if these interfaces are not works of art, then it begs the question “what is the role of the collaborative interface within a musical performance?”. Are they instruments in and of themselves? Are they improvisational compositions similar to Cage's *Musicircus*? Perhaps something in between? As part of his “frameworks and affordances” model, Mooney (2011) separates tools for music-making into two categories; physical and conceptual frameworks. He defines a framework as “any entity, construct, system, or paradigm – conceptual or physical – that contributes in some way to the composition or performance of music” and “any single framework can also be regarded as a collection of independent, smaller, frameworks”. Physical frameworks are the interactional tools used to create or compose music, such as instruments and manuscript paper, as well as recording equipment and music software like Sibelius or SuperCollider. Conceptual frameworks are the intangible tools used to structure music and performance, such as notation, scales, and free improvisation. Contextualising the practice of live coding using this model, we can define text editors and live coding languages separately as physical and conceptual frameworks respectively, but we can also combine these into the singular entity of the “live coding interface”. We could even go as far as to include the keyboard and mouse as physical frameworks that contribute to the live coding interface framework. The affordances of a framework exist along a spectrum of how easy they are to actualise by the user. For example, a live coding interface typically affords the ability to type characters onto the screen but it is more difficult to arrange those characters into valid syntax

for generating music. Harder still is the ability to collaborate with another performer without the addition of some networking functionality to the interface. A collaborative interface offers such additional affordances by embedding network communication within its core functionality. Highlighted by Figure 9.2, the process of ‘sharing code and/or data’ becomes a much easier task when using a collaborative live coding interface. Arguably, it makes it easier to share information than actually generate music using code. As a consequence, this unlocks other conceptual frameworks, such as group creativity, that enable creative interaction with other musicians through sharing information that would be near impossible otherwise. A framework will often leave a “characteristic fingerprint on the musical output” but does not predetermine it. Instead it provides a spectrum of possibilities that can be explored through creative practice and the role of the collaborative interface in performance is to extend these capabilities in an effort to enrich musical performance.

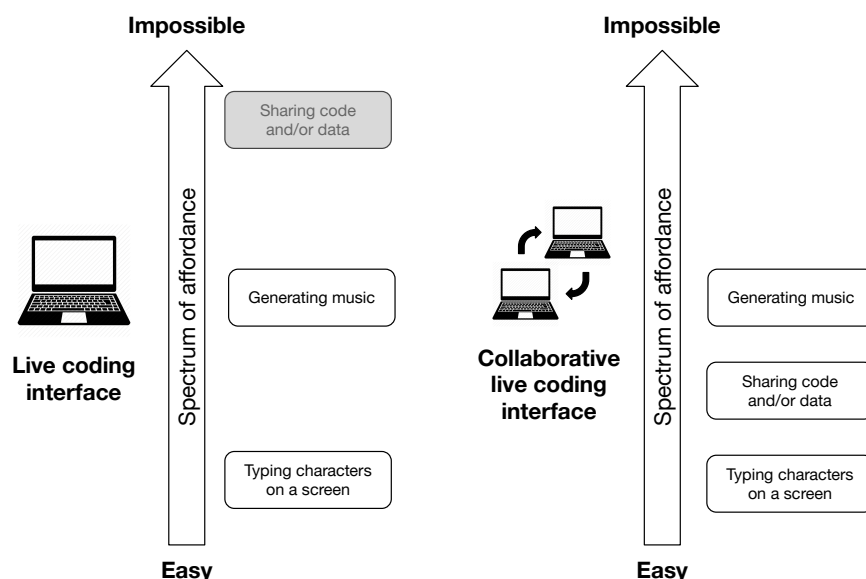


Figure 9.2: Spectrum of affordance for a live coding interface framework and a collaborative live coding interface framework. Adapted from (Mooney, 2011).

9.4 Addressing the Research Questions

1. How can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?

Before answering this question, the notion of what “better facilitated collaboration” is must first be addressed. As has been discussed throughout this thesis, Gifford et al. (2017) provides a framework for evaluating digital interfaces for creative improvisation, which includes assessing how well an interface enables flow, provides a sense of immediacy, and allows its users to take risks. To evaluate a digital interface for facilitating collaborative improvisation the same properties have

been examined but in the context of how well they are achieved by a group of users. Good collaboration occurs when performers are able to experience a sense of group flow, immediacy in communication as well as human-computer interaction, and trust in both their co-performers and the interface to take creative risks in live performance.

By allowing users to write code together and also communicate directly within a single, shared text buffer, the Troop interface facilitated collaboration at a deep and meaningful level. As one survey respondent (A4) said, “I think about things differently when collaborating with others – with troop you can see how other people are developing the code which in turn influences the decisions I make”. Being able to see co-performers’ coding activity is an important aspect of TYPE’s own performance practice, as Lucy mentioned in Section 5.6.2, it allows her to “feed off” her co-performers and gives her visual information to make informed decisions about what musical element to change. Furthermore, working on the same body of code as a group allows for errors to be addressed faster using “four pairs of hands”, as Laurie said (p. 67).

Troop’s shared console also notifies all ensemble members of error messages immediately. To resolve errors while working independently, live coders would have to communicate technical information verbally, which can be an arduous process. By sharing both code and information about the error, Troop gives users immediate access to the tools required to fix errors and continue with a performance. Dealing with errors is an important part of live coding practice and Lucy believes “it’s easier to stay in the flow and to keep things going with Troop compared to playing solo” because of the trust she has in her co-performers to “help fix it and recover” (p. 67). This trust was built up over a period of months and was mostly achieved through regular rehearsals using the Troop software. Troop has been extremely beneficial to organising these rehearsals too; the ability to play together over the internet has enabled the ensemble to rehearse more frequently and develop a level of trust and synergy much quicker than if they had had to rehearse in person every time.

Where Troop helped facilitate live coding collaboration through the medium of text, FoxDot was also able to enable collaborative practice through the medium of data. The FoxDot player-key syntax simplified and accelerated data-based collaboration, such as sharing patterns of numbers, between performers using Troop. While our synergy as an ensemble improved through regular rehearsals, using the player-key made the process of collaboration “easier” and helped create more “unified” and “coherent” music (p. 98), as discussed in Section 6.5.2. During the development of FoxDot’s collaborative syntax features it became clear that programming languages are not the most intuitive interfaces. Using representations of real-world actions, such as pitch accompaniment, were more useful in practice than theoretical programming constructs, such as lambda functions, even though they offered more powerful and flexible control over musical sequences. When more complex syntax *was* utilised during performance, it was only ever done so by a small subset of the

ensemble, which was deemed “less collaborative” by the others (p. 100). Lucy also felt that using simple code was “more important when you’re playing in a group because you need to be much more tuned in to what everyone else is doing” (p. 100). Being able to see your co-performers’ code is one thing but it is also important to understand it in order to use that information to inform your own decisions. Using simple and easy-to-remember syntax, like the `player-key` data type, also helped improve performers’ confidence in taking risks and reduced the time required to “translate” musical ideas into code (p. 101), which also gave performers a better sense of immediacy in the software.

It could be argued, however, that using simple syntax lowered the level of skill on display during performance as the members of the ensemble were now showcasing a breadth of technical programming knowledge. Creativity, in any form, is often associated with a high level of skill, but members of TYPE demonstrated their creative utility on a number of occasions despite embracing more simplistic linguistic features. In 1969, the Scratch Orchestra was created and brought together musicians of varying levels of skill (Cardew, 1969). In fact, the level of skill was so varied that they did not refer to themselves as musicians, but as “enthusiasts” and their concept of music was not limited to the purely auditory domain but was referred to “flexible and depends entirely on the members”. Like simplifying FoxDot’s syntax, this was designed to make the shared musical experience as inclusive and open as possible. For the Scratch Orchestra, it was a “Utopian vision of open enquiry and unfettered exploration, of an all-inclusive form of social music-making and performance” (Parsons, 2001). The Scratch Orchestra also demonstrated how creative musical performance could take place despite the varying level of skill among the group. They would often play “popular classics”, including music by composers such as Mozart and Brahms, but with only one performer knowing the full piece. The remainder of the orchestra would play along to the best of their abilities and fill in any unknown sections with improvised material and “variation arose naturally from differences of ability and from the wide disparity between intentions and results”. In a similar vein to the Scratch Orchestra, the Portsmouth Sinfonia was created in 1970 by a group of students at the Portsmouth School of Art where the only criteria for joining was that you didn’t know how to play your instrument. The idea was that the music they played would be about coming together and just enjoying the process. As a result, attempted performances of classical pieces of music became unique interpretations filled with dissonance and uncertainty. Brian Eno, who performed as part of the orchestra, said that this range of ability created great musical variety but that it was “achieved not by people trying to do something different from one another, but by accidentally doing something different; this sense of a limitation being turned into a strength” (Cairns, 2004). Despite a disparity in the levels of skill and knowledge in TYPE, performers were also able to open new avenues for creativity by exploring code in ways it was perhaps not designed for; Laurie mapping amplitude and “echo”, Innocent using the `accompany` syntax on a

whole chord, and Lucy doubling and halving pitch values for example. Constraints can often drive creativity (Stokes, 2005) and technical limitations can force artists to find innovative ways to make music (White, 2019). Using simple syntax not only strengthened the feeling of collaboration during performance, but also enabled creative utility of the code despite a range in the level of skill among the ensemble.

Chapter 7 saw the introduction of a new collaborative interface, CodeBank, which was more complex in its set-up than Troop. The complexity was, in part, created by the separation of commands into independent parts of the interface, such as the text editor, shared code, and chat features, which were all handled simultaneously within the shared text buffer in Troop. Giving users a private text editor to test out their ideas gave them more confidence but the overall code development was not of the same collaborative nature as it was when we had used Troop. Lucy said that, with CodeBank, “I’m more likely to react to something you do by changing something I’ve done” (p. 126). Furthermore, private working in ensemble live coding did not foster the same sense of familiarity and trust between performers as working with Troop, which also made it difficult to edit co-performers’ code. Given the time and privacy to develop one’s own code in CodeBank, performers also felt that there was “an expectation of perfection” and that “being able to push out something that’s a lot more developed means that there’s less room for other people to change that” (p. 128). As a consequence, the music did not develop very “organically”, but this striving for perfection did lead to the creation of well polished pieces of music. This raises the question; what is the goal of the ensemble? To achieve “peak jamming” (Swift, 2013) or to create a high-quality piece of music? This research is primarily concerned with the former but a system like CodeBank gives ensembles the choice of focusing on achieving group flow in improvisation or creating highly-polished musical performances.

The separation of many of Troop’s features into separate aspects of the interface seemed to increase the cognitive load on the performers who felt that they were “focusing on the code so much” (p. 128) that they were incognisant of chat messages and the content of the public codelet repository. Without a clear overview of the entire system state, including explicit chat messages, users lacked the awareness of the rest of the ensemble to achieve group flow. Users tended to enter states of personal flow in this method of working but found that not only were they unaware of chat messages and changes to codelets, but they would also be easily disrupted from this state as soon as they encountered an error. Unlike with Troop, this error usually had to be dealt with independently, which meant it would often take longer to resolve. Users also felt that communication in Troop was “a lot better and a lot more immediate” (p. 129) as they could combine code and communication into one action, which helped fix errors, demonstrate new ideas, and also send informal messages. It also appears that more functionality does not always equate to a better program; CodeBank introduced several new live coding functionalities, such as “rolling back” changes made to codelets,

that were accessible through button presses but were rarely utilised during performances. Even though they provided users with novel types of interactions, they were not intuitive to the practice that had developed as an ensemble through using Troop.

Collaborating with Troop made achieving group flow easier than it did with CodeBank as it provided a greater sense of immediacy when using the interface. The final project, Polyglot, attempted to emulate this but across a language barrier by combining multiple shared text buffers into a single window, each connected to a different live coding language. While this system was limited by several technological constraints, it did show promise as a potential tool for showcasing multilingual live coding virtuosity. Just as complex FoxDot syntax was detrimental to collaboration in Troop, including an unknown language's syntax in the editor made it "difficult to create a new line of code" (p. 147) that would compliment it as there was no way of knowing what the unknown language's code would sound like. One of the most consistent themes across these interfaces has been the reliance on visual information of the system state to achieve group flow. Seemingly an essential aspect of the practice, members of TYPE have found that being able to see and understand one another's code assists them in making creative decisions during performance. However, even if all performers using Polyglot could understand all of the languages being used, there are still issues that arise as a result of multi-lingual live coding; they don't inherently share the same "sonic character" (p. 147), which can make it difficult to write code in one language to compliment code in another. This was, to some extent, addressed in a rehearsal in which the same SynthDef was being triggered in both FoxDot and TidalCycles, and edited at its source in SuperCollider. As Lucy says, this opens a "huge amount of potential" (p. 148) for cross-language collaboration at the deep and meaningful level that is currently facilitated through the Troop interface. It would have been interesting to see what tacit knowledge might have been developed had there been more opportunities to explore the relationship between sound source manipulation and the coding of rhythm and pitch using Polyglot in this way. Unfortunately, Polyglot's use of multiple text buffers does create problems; users' attentions are divided in a similar way to how they were when using CodeBank. Even informal chat messages are ignored if users are not working in the same text buffer. As soon as the system state is divided into multiple parts, the cognitive load required to keep track of everything is multiplied and impedes the ensemble's ability to achieve a group flow-state. Furthermore, there still exist technical issues with the Polyglot interface that relate to performance on low powered computers. Its poor responsiveness during performance hindered the ensemble in establishing any sort of flow, group or individual, and also removed any sense of immediacy in the ensemble communication via text.

Many of the tools that helped facilitate collaboration in our own performances emerged through the tacit knowledge developed over time using the interfaces "in the wild". The combination of this craft knowledge and the iterative and reflexive methodology of participatory design formed a

symbiotic relationship that produced better software for collaborative live coding. By reflecting on practice and identifying the emerging themes and strategies it was possible to develop new iterations of software that attempted to adapt to performers' implicit needs. In Section 7.4, for example, I added keyboard shortcuts to the CodeBank interface to allow users to push code to the public workspace without having to use the mouse to click a button as this was the learned behaviour of the users. The cognition required to break the muscle-memory habit of navigating an interface with the keyboard was enough to disrupt a user's sense of flow, which is key to improvised musical performance. The question, "how can collaboration in ensemble live coding be better facilitated through performance systems, such as language, and interface design?", may not be completely answered, but through the cyclical process of reflection and software development this research has provided several insights into how it can be achieved. I have found that users need total access to the system state, including all of the code and any explicit written communication. Separating these divides users' attention and inhibits group flow, but it is better to do this than not provide all of the possible information. The system state also includes all of the user's current and previous actions; this can be well represented through colour association and should be made explicit wherever possible. Performances were also successful when everything was as simple as possible. This included set-up, syntax, as well as functionality. A simple set-up that allows users to connect and play together over the internet enables more frequent rehearsal time and helps develop trust and synergy amongst the ensemble. Simple syntax gives all members of the ensemble a level playing field with regards to their technical ability and minimises the time it takes to translate musical ideas into code on a screen. Simple functionality, or at least functionality that does not over-extend past users' existing muscle-memory, can help users learn systems more easily and minimise the cognitive work required to adapt to new processes. Simplifying the software itself can also help reduce performance issues on lower powered computers which can otherwise be a source of frustration and distraction, inhibiting an ensemble from achieving group flow.

2. How are collaborative interfaces used to reveal the creative processes at play in ensemble live coding performance?

This research question was posited in response to the lack of ensemble interaction made explicit by the majority of existing collaborative live coding interfaces. By developing interfaces with the aim of addressing this issue, how would creative processes be revealed in performances throughout this study? What does "revealing creative processes" even mean in this context? Is it something that occurs by design in the interface that highlights aspects of a live coding performance that would otherwise be invisible? This is how I tended to address this question when designing collaborative interfaces; asking myself how I could make these processes clearer to both performers and audiences alike. However, there were also unexpected methods for performer interaction that

occurred throughout this process that not only translated traditional ensemble interaction into a digital medium but also created new ways to interact with those experiencing the performance. These examples of creative processes being revealed were the result of the learned tacit knowledge that came from *using* the software, not just testing it.

Even as early as the initial rehearsal sessions using Troop there were instances of what I describe as “signposting” in the code that helped guide co-performers to the source of certain sounds or errors. In writing, signposting is the act of guiding your reader through a static document using structured sections and connecting phrases as the reader navigates the work from left to right, top to bottom. In live coding, however, the document is in a dynamic state of flux and its structure does not reflect the chronology of which it was written. To signpost for the readers, i.e. co-performers and members of the audience, TYPE would use code comments in a spatial dimension to draw attention to certain sections of code. Early on in our time as an ensemble this was often used to ask questions about syntax or errors but as the skill level of the group increased we continued to use comments in a more conversational, and even evaluative, way. Where members of a jazz ensemble might nod in approval in the direction of their co-performer who just produced a particularly great solo, we found ourselves adding comments onto the ends of lines of code that complimented that particular addition, as seen in Figure 9.3a for example. The use of comments also became quite performative over time; directed to the audience as much as they were to the rest of the ensemble. Figure 9.3b shows part of a screenshot from the Algorave Assembly lunch time concert in which I float the idea of moving onto a percussive section and Laurie responds with good humour, which translated well to the audience as it garnered some laughter. There were even times where lines that only contained comments were evaluated repeatedly in order to flood the console with text and bring attention to crucial information such as the amount of time remaining in a performance. This is a great example of “misuse” of an interface and epitomises the tacit knowledge accrued through real-world use of the software in a prolonged ethnographic study. Comments are features of computer code that are intended for humans only (they are discarded at the first stage of compilation into binary computer code) so to the computer, evaluating a line of comments does absolutely nothing but to those looking at the interface it communicates information with a sense of urgency and importance. This was not a behaviour that was imparted on the interface through its design but one that emerged through use in the hands of a real ensemble performing and rehearsing together over time.

When performing with CodeBank, which didn’t use a traditional a shared text buffer, it became increasingly difficult to signpost the code. CodeBank separated the code and the comments into different sections of the interface and, as noted in Section 7.6, was described as obfuscating creative processes as opposed to revealing them. Unlike Troop and Polyglot, CodeBank did make use of spatiotemporal relationships, i.e. the occurrence of an action related to its location in the interface.

```

1,1], sus=1, drive=0.5) + Pvar([0,[0,0,0,3,5]]
3,8],8), amp=1.25, lpf=4000, hpf=linvar([0,100
Ryan
shape=.4, ch).spread() # ooooh nice
inno lucy
an=(-1,1), room=0.5, mix=0.5, echo=PRand([0,0.

```

(a)

```

l3 >> play("uu ii", dur=2, amp=.6, rate=PRand(
# Percussive section? percussive section.
loz
Root.default=var([0,2,4],32)
l1 >> prophet(var([0,6],[2,1]), oct=PStep(5,4,
an=z1.pan, room=[ryan], chop=var([3,6],[12,5])
).spread().stop()
p1 >> k1ank(oct=var([4,5,3],[10,4,2]), chop=4,

```

(b)

Figure 9.3: Examples of comments being used during performance.

The most recently edited code would be at the top of the public repository and the most recently added message would be at the bottom of the chat log. However, the difference in frequency of changes to either part of the interface would mean that the most recent chat message would, more often than not, be out of date or unrelated to the most recent code changes. As mentioned above, the spatial signposting using comments had become part of our performance practice itself, not just a feature of the interface, and by removing it we lost access to an important means of revealing our creative processes to both the audience and to each other.

With this in mind, a possible extension of the Troop and Polyglot interfaces would be to add a temporal relationship to the code by highlighting each character as it is entered then fading the highlight out over time. It has already been demonstrated in Section 5.5.1 that the font colours of these interfaces can be augmented to change over time and by adding this behaviour at the individual character level we can make use of both space and time to share more information and make more informed decisions during a performance. While some aspects of the design of the interfaces resulted in the obfuscation of creative processes, there were several elements of their designs that helped reveal them. Troop, for example, did so very effectively; performers could see and work directly on their co-performers' textual material, which was made very visible to the audience. Working on the same body of code also enabled performers to combine together for new and exciting modes of ensemble interaction. For example, during the Algorave Assembly performance, described in Section 5.4.2, there were several moments in which performers came together within the code to create rich and satisfying musical combinations. Using colours to denote the author of each character highlights these combinations to the audience that would not otherwise be possible if not for Troop's functionality. One of the downsides to using text colour in this way, however, is that it becomes very difficult to implement any sort of syntax highlighting, which can make writing syntactically correct code difficult at times.

Syntax complexity also plays a role in revealing creative processes in ensemble live coding. Complex syntax creates a language barrier that even exists within an ensemble of relatively experienced live coders; as Lucy said, if someone writes "a really complicated line of code then the rest of us don't touch it" (p. 100). Ensemble interaction can only be seen by an audience if it is

actually occurring and using varying levels of technically complex code can impede this. TYPE felt that using simple code facilitated our interactions better and helped develop synergy within the group; “the energy works a bit better if it’s simple” 100, and this has translated well to audiences during performances. The CodeBank interface separated ensemble live coding into public and private activities such that audiences could only see and hear “finished” codelets and were not privy to their development in performers’ private workspaces. Audiences were not able to see any sort of history of a codelet, just the identity of the last performer to update it, which meant there was no evidence of interaction displayed on screen. The creative processes were actually *obscured* as opposed to revealed by the CodeBank system. As mentioned above, the separation of informal chat messages and code into separate features of the interface also impeding the ensemble’s ability to signpost the performance with interactional text and performative comments, further restricting creative processes from revealing themselves. The use of code comments as a medium for communication is another example of the tacit knowledge developed using Troop; the de facto method of communicating within the code did not happen by design but it became an integral part of the practice. If we consider this style of communication to be “craft knowledge” then perhaps the transition to a chat-box style of conversation was just “different” as opposed to worse. In either case, these differences do provide evidence that the design of an interface can affect how ensemble interaction is demonstrated to audiences in live coding performance. Regardless of these issues, however, giving performers a private text box to work in did yield highly polished musical results, as one audience member said, “it sounded like you knew what you were doing – everything gelled”. Like the Extramuros collaborative live coding system, Polyglot uses multiple text buffers within a single display but, unlike Extramuros, they are separated based on language instead of user. Combining all of the code within a single screen only requires a single projector for it to be shown to the audience and helps alleviate any problems associated with using multiple projectors. Like Troop, Polyglot uses coloured fonts and name-tags that are consistent across text boxes to help audiences identify performers’ code and follow their interactions, regardless of which text box they have used. Separating the code into more than one text box is unavoidable when using multiple languages simultaneously but by giving users an identifiable coloured font and name-tag, ensemble interactions in the code are made visible to the audience. This also allows users to signpost code relative to its location, similar to Troop, even if it is not a language they are using themselves. However, the addition of text buffers for languages that were not known to some performers made inter-language collaboration quite difficult. Innocent said that not being able to understand code written in TidalCycles was a “barrier” for him (p. 147) as he would often use the syntax or values of other FoxDot code when writing something to accompany it. By combining multiple languages into a single interface, creative processes can become obscured between ensemble members as it is not always possible to decipher meaning from one language to another. The ability (or perhaps

requirement) to not only see but to understand co-performers' code in order to collaborate effectively likely became ingrained in our practice as a result of performing together for such a long time using the same language across the whole ensemble. This sort of tacit knowledge can take time to un-learn or adapt and Polyglot would have benefited from longer rehearsal periods and several more performances "in the wild". Better yet, it would be useful to bring together a variety of live coders who specialised in each of the FoxDot, TidalCycles, and SuperCollider languages. This would provide a useful insight into the effect of real-time visual information about collaborator's code on performance and would also create a novel audience experience.

9.5 Conclusion

Through a series of iterative and reflexive software design cycles I have produced four interfaces for collaborative live coding including the extension of a live coding language. Each interface explores various facets of ensemble communication within the context of live coding, which has the novel characteristic of being primarily mediated through text, with the aim of better facilitating group collaboration and revealing creative processes to performers and audiences alike. I have used a participatory design methodology to develop these interfaces, which focused on working with users over long periods of time in an attempt to better understand performers' tacit knowledge and encapsulate it within the software's design. I have also embedded myself within this project as a practitioner-researcher, playing both the roles of designer and user, to not only better understand the process of collaborative live coding but also to develop my skills as a live coding musician. This provided me with the invaluable experience of being able to understand the ensemble's tacit knowledge of the systems that would otherwise be difficult to communicate if I was acting as a third-party observer. The evaluation of these designs combined autoethnographic reflections and observational studies of performances recorded "in-the-wild", as well as group interviews with the TYPE ensemble over the course of a two year period. Included in this thesis are twelve recordings from rehearsals and performance that took place during this time that showcase a portfolio of musical styles ranging from algorithmic dance music to experimental noise music. Through these performances we developed tacit knowledge in how to improvise together and use our interfaces to collaborate effectively without commonly used techniques for ensemble communication such as eye contact and body language. This knowledge, which I sometimes refer to as performance style or practice, was used as a guide to improve each iteration of the interface's design. As well as developing my artistic practice, I have also gained technical skills in software development and creating network applications for collaborative music performance, which I hope to continue building on in the future in a creative capacity.

Several themes emerged as a result of this work that illustrated key aspects of the ensemble's

interaction with the software. Firstly, there was a trade-off between an interface's complexity and the ability to achieve a sense of group flow while using. Simplicity in design fostered a greater sense of group flow as it minimised the cognitive load on the performers and allowed them to cognisant of their co-performer's actions while also focusing on on their own musical output. Simplifying the coding language itself also gave the ensemble members a "level playing field" such that each performer could understand – and be understood by – the others. The interpersonal chemistry of the group improved as a direct result of this. It also reduced the time spent deciphering code, freeing performers up to concentrate on creating music as opposed to interpreting it. Like with many constraints, using simplified syntax encouraged the group to come up with new and innovative ways of writing code. In doing so the group would often subvert the language's intended use, adding an element of indeterminacy and experimentation to a performance.

Another pattern of behaviour that emerged throughout this study was the "signposting" of code; adding comments with a spatial or temporal relationship to specific parts of the code to help both the co-performers and audience members navigate the dynamic-natured text. What started as a useful method for learning (questions and demonstrations) grew into a conversational and performative act that became idiosyncratic of our practice. Spatial signposting proved a key facet to our performance practice as, without it, the ensemble struggled with a crisis of identity and it became difficult to share creative processes with each other and the audience.

The ability to rehearse and perform telematically also proved to be an important part of the ensemble's musical development. Using the internet to facilitate rehearsals enabled the group to practice together more frequently as it was much easier to co-ordinate schedules without incorporating travel times or rehearsal space. The high frequency of rehearsal sessions led to the rapid development of trust and synergy among the group, which are key factors for successful improvisation. By using interfaces that were able replicate the environment for performing apart as it they are together, the ensemble were also able to rehearse in a way that would translate effectively to a live, co-located performance. By contrast, the ensemble found having to rehearse using a system that required performers being co-located (CodeBank) quite problematic. Living in separate cities essentially only allowed the group to rehearse on weekends and this was not possible on a weekly basis. As a result less time was spent practising with CodeBank and the ensemble was not able to develop the same level of chemistry nor virtuosity when performing with it. Strong telematic functionality also proved valuable when scheduling gigs overseas as it allowed the ensemble to perform with only a subset of its members in the venue. Our society is increasingly experiencing life online and remote musical performance could well play a part in its future.

One thing that has not been mentioned over the course of this thesis is the profound effect this research has had on my social well-being. As well as producing several pieces of software for musical collaboration I have also developed some excellent friendships that have transcended the

musical ensemble dynamic. Lucy, Laurie, and Innocent have not only become my collaborators and helped me broaden my musical horizons, but have also become my close personal friends. The long and arduous journey of the PhD student is typically a solitary one and having people to rely on and support you throughout the process has made it feel much more achievable. I had been working on FoxDot and performing as a live coder independently for some time but doing it with friends at my side breathed new life into the practice for me. I would probably argue that, out of all of the things that have happened as a result of this research, this has been the best.

References

- Aaron, S. (2016). Sonic pi – performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2), 171–178.
- Aaron, S., Blackwell, A. F., Hoadley, R., & Regan, T. (2011). A principled approach to developing new languages for live coding. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 381–386). doi: 10.5281/zenodo.1177935
- Armitage, J. L. (2017). *Portfolio of original compositions with written commentary* (Unpublished doctoral dissertation). University of Leeds.
- Auslander, P. (2008). *Liveness: Performance in a mediatized culture*. London: Routledge.
- Bailey, D. (1992). *Improvisation: its nature and practice in music* (2nd ed.). London: British Library National Sound Archive.
- Barbosa, Á. M. M. (2003). Displaced soundscapes: A survey of network systems for music and sonic art creation. *Leonardo Music Journal*, 13, 53–59. doi: 10.1162/096112104322750791
- Barbosa, Á. M. M. (2006). *Computer-supported cooperative work for music application* (Unpublished doctoral dissertation). Universitat Pompeu Fabra.
- Bartleet, B.-L. (2009). Behind the baton: Exploring autoethnographic writing in a musical context. *Journal of Contemporary Ethnography*, 38(6), 713–733.
- Baumann, T. (2015). *Operational transformation - OT explained*. <http://operational-transformation.github.io/>. (accessed 23/04/18)
- Bekhet, A. K., & Zauszniewski, J. A. (2012). Methodological triangulation: An approach to understanding data. *Nurse researcher*, 20(2).
- Benford, S., Greenhalgh, C., Crabtree, A., Flintham, M., Walker, B., Marshall, J., . . . Row Farr, J. (2013). Performance-led research in the wild. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 20(3), 14.
- Bird, D. (2010). *Fields*. <http://davidbird.tv/new-page-2>, accessed 26/11/18.
- Bischoff, J., Gold, R., & Horton, J. (1978). Music for an interactive network of microcomputers. *Computer Music Journal*, 2(3), 24–29. doi: 10.2307/3679453
- Blackwell, A., & Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of psychology of programming interest group* (pp. 284–289). Brighton, United Kingdom.
- Blackwell, A., McLean, A., Noble, J., & Rohrerhuber, J. (2014). Collaboration and learning through live coding (dagstuhl seminar 13382). *Dagstuhl Reports*, 3(9).
- Borgdorff, H. (2010). The production of knowledge in artistic research. In M. Biggs & H. Karlsson

- (Eds.), *The Routledge companion to research in the arts* (pp. 74–93). Abingdon: Routledge.
- Brand, G., Sloboda, J., Saul, B., & Hathaway, M. (2012). The reciprocal relationship between jazz musicians and audiences in live performances: A pilot qualitative study. *Psychology of Music, 40*(5), 634–651.
- Brown, A. R., & Sorensen, A. C. (2007). aa-cell in practice: An approach to musical live coding. In *Proceedings of the international computer music conference* (pp. 292–299). Copenhagen, Denmark.
- Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: a way to learn programming principles. *Education and information technologies, 2*(1), 65–83.
- Burland, K., & McLean, A. (2016). Understanding live coding events. *International Journal of Performance Arts and Digital Media, 12*(2), 139–151.
- Burland, K., & Pitts, S. (2012). Rules and expectations of jazz gigs. *Social Semiotics, 22*(5), 523–543.
- Børkder, S., Grønbaek, K., & Kyng, M. (1995). Cooperative design: techniques and experiences from the scandinavian scene. In *Readings in human-computer interaction* (pp. 215–224). San Francisco: Morgan Kaufmann. doi: 10.1016/B978-0-08-051574-8.50025-X
- Cage, J. (1967). *Musicircus*.
- Cairns, D. (2004). *Portsmouth sinfonia*. <https://www.portsmouthsinfonia.com/media/sundaytimes.html>. (accessed 01/11/19)
- Cardew, C. (1969). A scratch orchestra: draft constitution. *The Musical Times, 110*(1516), 617–619.
- Cascone, K. (2000). The aesthetics of failure: “post-digital” tendencies in contemporary computer music. *Computer Music Journal, 24*(4), 12–18. doi: 10.1162/014892600559489
- Cascone, K. (2011). *Errormancy: Glitch as divination*. <http://www.theendofbeing.com/2012/04/19/errormancy-glitch-as-divination-a-new-essay-by-kim-cascone/>. (accessed 24/11/2016)
- Chin, J. P., Diehl, V. A., & Norman, K. L. (1988). Development of an instrument measuring user satisfaction of the human-computer interface. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 213–218). doi: 10.1145/57167.57203
- Clift, S., Hancox, G., Morrison, I., Hess, B., Kreutz, G., & Stewart, D. (2010). Choral singing and psychological wellbeing: Quantitative and qualitative findings from english choirs in a cross-national survey. *Journal of Applied Arts & Health, 1*(1), 19–34.
- Cockos Incorporated. (2018). *Cockos incorporated — ninjam*. <https://www.cockos.com/ninjam/>.
- Collins, N., McLean, A., Rohrhuber, J., & Ward, A. (2003). Live coding in laptop performance. *Organised sound, 8*(03), 321–330.

- Courtney, D. (1994). The cyclic form in north indian tabla. *Percussive Notes*, 33(6), 32–45.
- Crawford, L. (1996). Personal ethnography. *Communications Monographs*, 63(2), 158–170.
- Csikszentmihaly, M. (1991). *Flow: The psychology of optimal experience*. New York, NY: Harper and Row.
- Cunningham, S. J., & Jones, M. (2005). Autoethnography: a tool for practice and education. In *Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on computer-human interaction: making chi natural* (pp. 1–8). doi: 10.1145/1073943.1073944
- de Campo, A. (2014). Republic: Collaborative live coding 2003–2013. In A. Blackwell, A. McLean, J. Noble, & J. Rohrhuber (Eds.), *Collaboration and learning through live coding (Dagstuhl seminar 13382)* (Vol. 3, p. 152-153).
- de Carvalho Junior, A. D., Lee, S. W., & Essl, G. (2015). Supercopair: Collaborative live coding on supercollider through the cloud. In *Proceedings of the first international conference on live coding* (pp. 152–158). doi: 10.5281/zenodo.19347
- Del Angel, L. N., Teixido, M., Ocelotl, E., Cotrina, I., & Ogborn, D. (2019). Bellacode: localized textual interfaces for live coding music. In *International conference on live coding* (pp. 27–36). Madrid, Spain.
- Dijkstra, E. W. (1960). Recursive programming. *Numerische Mathematik*, 2(1), 312–318.
- Eldridge, A., & Kiefer, C. (2017). The self-resonating feedback cello: interfacing gestural and generative processes in improvised performance. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 25–29). doi: 10.5281/zenodo.1176157
- Ellis, C. A., & Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on management of data* (pp. 399–407).
- Emmerson, S. (2007). *Living electronic music*. Aldershot: Ashgate Publishing, Ltd.
- Fencott, R. (2012). *Computer musicking: Designing for collaborative digital musical interaction*. (Unpublished doctoral dissertation). Queen Mary University of London.
- Fencott, R., & Bryan-Kinns, N. (2010). Hey man, you're invading my personal space! privacy and awareness in collaborative music. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 198–203). doi: 10.5281/zenodo.1177763
- Fencott, R., & Bryan-Kinns, N. (2013). Computer musicking: HCI, CSCW and collaborative digital musical interaction. In *Music and human-computer interaction* (pp. 189–205). London: Springer.
- Flašar, M. (2016). Listening with the eyes: Remarks on live coding performance. *Media | Archive | Performance*, 7. <http://www.perfomap.de/map7/media-performance-on-gestures/listening-with-the-eyes-remarks-on-live-coding-performance>. (accessed 24/11/16).
- Flor, N. V. (2006). Globally distributed software development and pair programming. *Communications of the ACM*, 49(10), 57–58.

- Freeman, J., & Van Troyer, A. (2011). Collaborative textual improvisation in a laptop ensemble. *Computer Music Journal*, 35(2), 8–21.
- Furniss, P. (2016). *Live coding meets augmented instruments*. <https://petefurniss.wordpress.com/2016/01/12/live-coding-meets-augmented-instruments/>. (accessed 07/05/19)
- Garland, Z. (2001). *BBC - h2g2 - oblique strategies*. <http://www.bbc.co.uk/dna/place-nireland/A635528>. (accessed 24/11/16)
- Geiger, C., Reckter, H., Paschke, D., Schulz, F., Poepel, C., & Ansbach, F. (2008). Towards participatory design and evaluation of theremin-based musical interfaces. In *Proceedings of the conference on new interfaces for musical expression* (pp. 303–306). doi: 10.5281/zenodo.1179545
- Gifford, T., Knotts, S., Kalonaris, S., & McCormack, J. (2017). Evaluating improvisational interfaces. In *Proceedings of the improvisational creativity workshop*. Prato, Italy.
- Google Inc. (2017). *Google docs - create and edit documents online, for free*. <https://www.google.co.uk/docs/about/>. (accessed 02/02/17)
- Google Inc. (2018). *G suite for education — google for education*. <https://edu.google.com/products/gsuite-for-education/>. (accessed 01/05/19)
- Goyvaerts, J. (2016). *Regex tutorial, examples and reference*. <https://www.regular-expressions.info/>. (accessed 12/12/18)
- Hugill, A. (2005). Internet music: An introduction. *Contemporary Music Review*, 24.
- Hummels, J. (2013). *Mind your own business — Jonas Hummel's archive*. <http://jonashummel.de/archives/projects/myob/>. (accessed 24/01/2018)
- Hutchins, C. C. (2015). Live patch / live code. In *Proceedings of the first international conference on live coding* (pp. 147–151). doi: 10.5281/zenodo.19346
- Janata, P., & Grafton, S. T. (2003). Swinging in the brain: shared neural substrates for behaviors related to sequencing and music. *Nature Neuroscience*, 6(7), 682–687.
- Johansson, M. (2009). *Instant music & messaging: Interconnecting music and messaging* (Unpublished doctoral dissertation). KTH Royal Institute of Technology.
- Kim-Boyle, D. (2009). Network musics: Play, engagement and the democratization of performance. *Contemporary Music Review*, 28(4-5), 363–375. doi: 10.1080/07494460903422198
- Kindler, E., & Krivy, I. (2011). Object-oriented simulation of systems with sophisticated control. *International Journal of General Systems*, 40(3), 313–343.
- Kirkbride, R. (2016). Foxdot: Live coding with python and supercollider. In *Proceedings of the international conference of live interfaces* (pp. 194–198). Hamilton, Canada.
- Knotts, S. (2016). Algorithmic interfaces for collaborative improvisation. In *Proceedings of international conference on live interfaces* (pp. 232–237). Brighton, United Kingdom.
- Knotts, S. (2018). *Social systems for improvisation in live computer music* (Unpublished doctoral

- dissertation). Durham University.
- Landry, S., & Jeon, M. (2017). Participatory design research methodologies: A case study in dancer sonification. In *International conference on auditory display* (pp. 182–187).
- Lazzaro, J., & Wawrzynek, J. (2001). A case for network musical performance. In *Proceedings of the 11th international workshop on network and operating systems support for digital audio and video* (pp. 157–166). doi: 10.1145/378344.378367
- Lee, S. W., & Essl, G. (2013). Live coding the mobile music instrument. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 493–498). doi: 10.5281/zenodo.1178592
- Lee, S. W., & Essl, G. (2014). Models and opportunities for networked live coding. In *Live coding and collaboration symposium*. Birmingham, United Kingdom.
- MacDonald, R. A. (2013). Music, health, and well-being: A review. *International journal of qualitative studies on health and well-being*, 8(1), 20635.
- Magnusson, T. (2011a). Algorithms as scores: Coding live music. *Leonardo Music Journal*, 21, 19–23. doi: 10.1162/LMJ_a_00056
- Magnusson, T. (2011b). ixi lang: a supercollider parasite for live coding. In *Proceedings of international computer music conference* (pp. 503–506). Huddersfield, United Kingdom.
- Magnusson, T. (2013). The threnoscope: A musical work for live coding performance. In *International workshop on live programming at the international conference on software engineering*. San Francisco, CA, United States of America.
- Magnusson, T. (2014). Herding cats: Observing live coding in the wild. *Computer Music Journal*, 38(1), 8–16. doi: 10.1162/COMJ_a_00216
- Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4), 16. doi: 10.1145/1868358.1868363
- Mann, M. (2012). *No talent is required to perform electronic dance music*. <https://www.straight.com/music/no-talent-required-perform-electronic-dance-music>. (accessed 05/04/19)
- McCaleb, M. (2011). Embodied knowledge in ensemble performance: The case of informed observation. In *Proceedings of the conference on performance studies*. Aveiro, Portugal.
- McCartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4), 61–68.
- McGrath, J. E. (1984). *Groups: Interaction and performance*. Englewood Cliffs NJ: Prentice-Hall.
- McLean, A. (2014). Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on functional art, music, modelling & design* (pp. 63–70). doi: 10.1145/2633638.2633647

- McLean, A. (2015). Reflections on live coding collaboration. In *Proceedings of the third conference on computation, communication, aesthetics and x.* (pp. 213–220). Porto, Portugal.
- McLean, A. (2017). *Live coding – POTAC – medium.* <https://medium.com/potac/live-coding-1eb06f0ddf26>. (accessed 19/01/17)
- McLean, A., Griffiths, D., Collins, N., & Wiggins, G. (2010). Visualisation of live code. In *Proceedings of electronic visualisation and the arts* (pp. 26–30). doi: 10.14236/ewic/EVA2010.6
- McLean, A., & Wiggins, G. (2010). Tidal-pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference* (p. 331–334). Barcelona, Spain.
- Merritt, T., Kow, W., Ng, C., McGee, K., & Wyse, L. (2010). Who makes what sound? supporting real-time musical improvisations of electroacoustic ensembles. In *Proceedings of the 22nd conference of the computer-human interaction special interest group of australia on computer-human interaction* (pp. 112–119). doi: 10.1145/1952222.1952245
- Mills, D., et al. (1985). *Network time protocol* (Tech. Rep.). RFC 958, M/A-COM Linkabit.
- Monson, I. (2009). *Saying something: Jazz improvisation and interaction*. Chicago, IL: University of Chicago Press.
- Mooney, J. (2011). Frameworks and affordances: Understanding the tools of music-making. *Journal of Music, Technology & Education*, 3(2-3), 141–154.
- Mori, G. (2015). Analysing live coding with ethnographic approach - a new perspective. In *Proceedings of the first international conference on live coding* (pp. 117–124). doi: 10.5281/zenodo.19343
- Norman, D. A. (1998). *The design of everyday things*. London: MIT Press.
- Nosnibor, C. (2016). *Review: 'Silver Apples'. headrow house, leeds, 24th august.* <http://www.whisperinandhollerin.com/reviews/review.asp?id=13148>. (accessed 08/12/16)
- Ogborn, D. (2012). Espgrid: a protocol for participatory electronic ensemble performance. In *Audio engineering society convention 133*. <http://www.aes.org/e-lib/browse.cfm?elib=16625>.
- Ogborn, D. (2018). Network music and the algorithmic ensemble. *The Oxford Handbook of Algorithmic Music*, 345–361.
- Ogborn, D., Beverley, J., del Angel, L. N., Tsabary, E., & McLean, A. (2017). Estuary: Browser-based collaborative projectional live coding of musical patterns. In *Proceedings of the international conference on live coding*. Morelia, Mexico.
- Ogborn, D., & Mativetsky, S. (2015). Very long cat: Zero-latency network music with live coding. In *Proceedings of the first international conference on live coding*. Zenodo.
- Ogborn, D., Tsabary, E., Jarvis, I., Cárdenas, A., & McLean, A. (2015). Extramuros: Making music in a browser-based, language-neutral collaborative live coding environment. In *Proceedings*

- of the first international conference on live coding (pp. 163–169). doi: 10.5281/zenodo.19349
- Oliveros, P. (2009). From telephone to high speed internet: A brief history of my tele-musical performances. *Leonardo Music Journal Online Supplement to LMJ*, 19, 2–5.
- Parsons, M. (2001). The scratch orchestra and visual arts. *Leonardo Music Journal*, 11, 5–11. doi: 10.1162/09611210152780601
- Polimeneas-Liontiris, T., Eldridge, A., Kiefer, C., & Magnusson, T. (2018). The ensemble as expanded interface sympoetic performance in the brain dead ensemble. In *Proceedings of the international conference on live interfaces*. (pp. 117–125). Porto, Portugal.
- Roberts, C., & Kuchera-Morin, J. (2012). Gibber: Live coding audio in the browser. In *Proceedings of the international computer music conference* (pp. 64–69). Ljubljana, Slovenia.
- Roberts, C., Yerkes, K., Bazo, D., Wright, M., & Kuchera-Morin, J. (2015). Sharing time and code in a browser-based live coding environment. In *Proceedings of the first international conference on live coding* (pp. 179–185). doi: 10.5281/zenodo.19351
- Rohrhuber, J., de Campo, A., Wieser, R., van Kampen, J.-K., Ho, E., & Hölzl, H. (2007). Purloined letters and distributed persons. In *Music in the global village conference*. Budapest, Hungary.
- Russolo, L. (1913). *The art of noise: futurist manifesto*. New York, NY: Something Else Press.
- Ruthmann, A., Heines, J. M., Greher, G. R., Laidler, P., & Saulters II, C. (2010). Teaching computational thinking through musical live coding in scratch. In *Proceedings of the 41st acm technical symposium on computer science education* (pp. 351–355). doi: 10.1145/1734263.1734384
- Sarwate, A., Rose, R. T., Freeman, J., & Armitage, J. (2018). Performance systems for live coders and non coders. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 370–373). doi: doi.org/10.5281/zenodo.1302627
- Saunders, J. (2015). *all voices are heard (2015) - james saunders*. <http://www.james-saunders.com/all-voices-are-heard-2015/>. (accessed 24/01/2018)
- Saunders, J. (2017). *What's the point? balancing purpose and play in rule-based compositions*. <http://www.james-saunders.com/whats-the-point-balancing-purpose-and-play-in-rule-based-compositions/>. (accessed 02/04/19)
- Sawyer, R. K. (2006). Group creativity: musical performance and collaboration. *Psychology of music*, 34(2), 148–165. doi: 10.1177/0305735606061850
- Sawyer, R. K. (2015). Group creativity: musical performance and collaboration. In R. Caines & A. Heble (Eds.), *The improvisation studies reader: Spontaneous acts* (p. 87-100). Abingdon: Routledge.
- Seddon, F. A. (2005). Modes of communication during jazz improvisation. *British Journal of Music Education*, 22(1), 47–61. doi: 10.1017/S0265051704005984
- Sicart, M. (2014). *Play matters*. Cambridge, MA: MIT Press.

- Sicchio, K. (2014). Hacking choreography: Dance and live coding. *Computer Music Journal*, 38(1), 31–39. doi: 10.1162/COMJ_a_00218
- Sorensen, A. C. (2010). A distributed memory for networked livecoding performance. In *Proceedings of the ICMC2010 international computer music conference* (pp. 530–533). New York, NY, United States of America.
- Sorensen, A. C. (2011). *Extempore*. <http://extempore.moso.com.au/>. (accessed 05/12/2016)
- Sorensen, A. C. (2013). *The many faces of a temporal recursion*. http://extempore.moso.com.au/temporal_recursion.html. (accessed 05/12/2016)
- Sorensen, A. C., & Gardner, H. (2010). Programming with time: Cyber-physical programming with impromptu. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications* (pp. 822–834). doi: 10.1145/1869459.1869526
- Spiegel, L. (1981). Manipulations of musical patterns. In *Proceedings of the symposium on small computers and the arts* (pp. 19–22). Philadelphia, PA, United States of America.
- Spinuzzi, C. (2005). The methodology of participatory design. *Technical communication*, 52(2), 163–174.
- Spry, T. (2016). *Body, paper, stage: writing and performing autoethnography*. Abingdon: Routledge.
- Stokes, P. D. (2005). *Creativity from constraints: The psychology of breakthrough*. New York: Springer.
- Swift, B. (2013). Chasing a feeling: Experience in computer supported jamming. In *Music and human-computer interaction* (pp. 85–99). Springer.
- Swift, B., Sorensen, A., Martin, M., & Gardner, H. (2014). Coding livecoding. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 1021–1024). doi: doi.org/10.1145/2556288.2557049
- TheCodingMonkeys GmbH. (2014). *SubEthaEdit4*. <https://www.codingmonkeys.de/subethaedit/>. (accessed 02/02/17)
- Thompson, R. (2008). The music of ryukyu. In A. Tokita & D. W. Hughes (Eds.), *The Ashgate research companion to japanese music* (p. 315). Aldershot: Ashgate Publishing, Ltd.
- TOPLAP. (n.d.). *Manifesto draft - TOPLAP*, note=accessed 08/12/16, year=2004,. <http://toplap.org/wiki/ManifestoDraft>.
- TOPLAP. (2004). *Historical performances - TOPLAP*. <https://toplap.org/wiki/HistoricalPerformances>. (accessed 05/04/19)
- Toussaint, G. T., et al. (2005). The euclidean algorithm generates traditional musical rhythms. In *Proceedings of BRIDGES: Mathematical connections in art, music and science* (pp. 47–56). Banff, Alberta, Canada.
- Valve Software. (2017). *Source multiplayer networking - valve developer community*.

- https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. (accessed 26/11/18)
- Veinberg, A., & Noriega, F. I. (2018). Coding with a piano: the first phase of the codeklavier's development. In *Proceedings of international computer music conference* (pp. 93–98). Daegu, South Korea.
- Veinberg, A., & Noriega, F. I. (2019). *ICLC 2019*. <http://iclc.livecodenetwork.org/2019/programa.html#pn8>. (accessed 07/05/2019)
- What Works Wellbeing. (2016). *Music, singing and wellbeing in healthy adults*. <https://whatworkswellbeing.files.wordpress.com/2016/11/wellbeing-singing-music-briefing-nov20162.pdf>. (accessed 12/10/2018)
- White, A. (2019). Analog algorithms: Generative composition in modular synthesis. In *Proceedings of the australasian computer music conference* (pp. 68–73). Melbourne, Australia.
- Williams, L. (2001). Integrating pair programming into a software development process. In *Proceedings 14th conference on software engineering education and training* (pp. 27–36). doi: 10.1109/CSEE.2001.913816
- Wilson, S., Lorway, N., Coull, R., Vasilakos, K., & Moyers, T. (2014). Free as in beer: Some explorations into structured improvisation using networked live-coding systems. *Computer Music Journal*, 38(1), 54–64.
- Wright, M., Freed, A., et al. (1997). Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 international computer music conference* (p. 10). Thessaloniki, Greece.
- Xambó, A., Freeman, J., Magerko, B., & Shah, P. (2016). Challenges and new directions for collaborative live coding in the classroom. In *Proceedings of the international conference of live interfaces* (p. 65-73). Brighton, United Kingdom.
- Xambó, A., Laney, R., Dobbyn, C., & Jorda, S. (2011). Collaborative music interaction on tabletops: an HCI approach. In *BCS HCI 2011 workshop on when words fail: What can music interaction tell us about hci?* Newcastle Upon Tyne, United Kingdom.
- Zmölnig, I. m. (2016). Audience perception of code. *International Journal of Performance Arts and Digital Media*, 12(2), 207–212.
- Zorn, J. (1987). *Cobra*. Basel, Switzerland: Hathut.

Appendix

Appendix A: Performance Descriptions

A.1 Leeds Algorave, Open Data Institute, Leeds - 28/04/17

Video recording: `ch5_1b-Leeds_Algorave-28_04_17.mp4`

(0:00 - 11:30) The first ten minutes of the performance centered around a bright melody and steady kick drum beat at 120 bpm. A constant feature of this performance was that majority of the code was written by myself in the green font. Troop's bar chart in the photograph in Figure 5.6 shows that I have written at least double the amount of code as my collaborators. This is not completely unexpected as I had been using the FoxDot language for a number of years whereas the rest of the ensemble had only started to learn it over the last few months.

(11:30 - 16:30) At the 12 minute mark we had reached a crescendo of sorts by increasing the density of the snare drum pattern, which used transformations of the Cuban cinquillo rhythm that is found in many modern Latin-American pop songs. It seemed this crescendo was felt by all performers as the scale was then changed and the drumbeat was removed by minute 13, leaving a barrage of electronic horn sounds. Percussion was gradually added back in and the horns were replaced with a ghostly synth stab and we returned to the bright melody and snare drum rhythm used to begin the set. The drum pattern was made more complex by applying more transformations to it and the bright melody was altered to create bursts of dissonance with a distorted bass synth added in the background.

(17:00 - 18:30) At this point you can see some discussion between the performers, with Laurie gesturing towards his screen to say that the text had become "scrambled". The error that had occurred in practice had now happened during a live performance. Due to the scrambled text, it was very difficult to update code effectively and the music had changed very little over the last few minutes. In an attempt to alter the overall soundscape I decided to change key and move it up two semitones. We tried to remove some elements from the music in order to get control over the situation, which involved stopping and deleting much of the code. I felt that this transition was well executed as the music did not change too abruptly and made a significant change in the musical direction. If anything the error was actually a positive thing as it became a driving factor for change in the music to recover the situation. Live coding is often described as the process of "embracing error" and letting failure lead you in musical performance and this moment was

definitely an example of this.

(20:00 - 28:40) We were not able to fully tame the error and at midway through minute 20 Laurie was standing with his hands on his hips looking exasperated because the text had become scrambled again. I made an executive decision at this point to start from scratch. I stopped all of the audio except a single bass synth note on repeat and gave it a varying high-pass filter effect in order for us to buy some time to delete the entire contents of the text buffer and regroup. We were still getting some errors in FoxDot so we made the decision to stop all of the audio, much to the delight of the audience. We wanted to end the performance on a bang and decided to increase the tempo. Again, this was a case of “embracing error” and using it as a driving force for musical change. The tone of the music after the break was much darker and closer to minimal techno than the Afro-Caribbean rhythms we were exploring previously. Similar to the first 10 minutes of the performance, we ramped up to a crescendo by adding more musical elements and utilizing dense snare drum rhythms. We finished by playing as many snare drums samples as we could before cutting all of the audio, creating a blast of noise, which seemed to be a crowd pleasing move.

A.2 Algorave Assembly Lunchtime Concert, Leeds - 27/04/18

Video recording: `ch5_2-Algorave_Assembly-27_04_18.avi`

(00:00 - 01:50) The first sound was created by Lucy who introduced a synth called “snick” which combines a haunting high-pitched tone and semi-rhythmic clicking sound. I responded by introducing a simple woodblock sequence with a repeated rhythm and heavy reverb filter to try and embellish the atmosphere. Laurie then added hi-hats and shakers, keeping in line with the higher frequencies and staccato rhythms already established. I develop the woodblock sequence by “stuttering” it every 7 beats but at twice the playback rate, further exploring the higher ranges of frequencies. This was not decided beforehand but, after rehearsing and performing together for just over a year, we had developed a chemistry that allowed us to instinctively react to each other’s music. Lucy then changed my percussive sequence by spacing out events and I edited the pitch of the “snick” synth to alter every 12 beats. Lucy then “stuttered” the sequence created by Laurie, creating a density to the sound. Being able to work within the same textual material provides you with the ability to not only easily replicate the techniques being used by your co-performers, but also affect them directly if you so wish. Without sharing our code via Troop, Lucy may not have known which functions I was using to create the sounds that I was (“stutter”) and may not have been able to easily reproduce the effect elsewhere in the code. This is a perfect example of how Troop gives performers complete holistic control over the performance as they are able to pick and choose aspects of the code they wish to alter.

(01:54 - 04:30) At the start of minute 2 I began to explore some lower frequencies for the “snick” synth and added some wave-shape distortion and a “chop” effect to see what would happen if we moved out the high pitched percussive section we had currently found ourselves in. The rhythm felt free and loose at this point and very much like a free improvisation; it had an almost human quality to it. Laurie then introduced a new synth, called “donk”, in a lower octave, seemingly following suit to my previous change. As we started to make more drastic changes to the mix I replaced the “snick” with a “prophet” synth while keeping all other aspects of the layer the same.

(04:30 - 05:50) Unfortunately, Laurie’s laptop then suddenly disconnected from the server, which was likely caused by a network drop. For Lucy and I, though, the program continued to work and we carried on while Laurie attempted to reconnect. At that point in the performance I felt that the sound was very centred around one tone and I decided to change the root note by moving it two semitones every 32 beats. Laurie managed to reconnect successfully but his font colour had changed from blue to green, and his previous contributions on the character-tracking graph

had gone. Lucy stopped and deleted one of the percussive sequences and Laurie increased the sustain value of the “donk” synth. Without realising it, we had entered into a synth heavy section and started to find a groove. This would often happen during rehearsals and performances; we compliment each other’s changes quite subconsciously when responding to both the music and the code in Troop. Being able to see, as well as hear, your co-performer’s work allows you to make predictions about the direction of the musical flow and also take inspiration for your own work.

(06:00 - 06:35) Laurie then increased the density of the “donk” rhythm further and, using varying octaves, also expanded the spectral variety. I introduced an ambient tone in a lower octave to flesh out the overall sound, perhaps subconsciously countering the last change made by Laurie, while he also added a kick drum to the percussive sequence. At this point, we began to overload the computer’s CPU, which caused SuperCollider to generate a glitching sound as it could not render audio correctly. Reducing some of the CPU intensive effects, such as reverb and delay, resolved the issue and we managed to move forward without any more trouble. Perhaps inspired by these gritty and glitched artefacts, distortion was added to the ambient tone, which created a contrast with the fluttering blips of the “donk” synth in the octaves above it. Once again our creative decisions during performance are being influenced by computer error. This is one of the beautiful things about live coding; it really is live and unscripted and you are never far away from a crash.

(06:45 - 09:40) I decided to add some repeated calls to to the “offadd” method in order to generate more notes from the “donk” synth; a layer that I was really enjoying at that point. It’s a very satisfying feeling when the combination of ideas - Laurie’s initial creation of the code and my own editing - contributes to a fresh musical sequence. I again changed the root note to move up another two semitones every 32 beats with the idea that this change in tonality might signal a greater musical change overall. Lucy eventually stopped the “donk” sequence, which made the soundscape suddenly feel very sparse. Lucy also changed the key to mixolydian and increased the tempo by 20 beats per minute; indeed, a change was coming. Lucy also introduced a snare drum based sequence that seemed to carry more urgency within the performance. I added distortion to the marimba in order to increase its prominence within the overall mix and it, too, began to carry a strong, dedicated rhythm. Laurie also updated the accompanying percussive sequence to match this steady rhythm and, once more, we ended up on the same page through the use musical and textual cues, as we entered a new phase in the performance.

(09:45 - 09:55) I wrote “percussive section?” in a comment, which was as much for the audience as it is my co-performers, and Laurie replied in the affirmative. This was received with a laugh from several members of the audience, and further indicated that a change in the music

was approaching. This interaction between Laurie and myself was facilitated by Troop and the audible response from the audience highlighted the fact that audiences are gaining insight into the ensemble’s communication through this tool.

(09:55 - 14:30) We removed all of synth tones within the soundscape and established a regular rhythm while still maintaining a sense of groove. I introduced a four-to-the-floor percussive sequence and utilised the “stutter” method to create an iterative sound gesture by pitch shifting the samples and Lucy responded by creating a sparse percussive rhythm of her own. As we developed our percussive sequences, Laurie began to write a long line of code for use with a synth called “zap”. Its introduction was subtle, but solid. It didn’t take anything away from the percussive background, which had been increasing in intensity as Lucy and I continued to embellish the sequences. Laurie’s “zap” sequence uses a TimeVar for its duration to achieve an interesting rhythmic effect.

(14:30 - 19:50) I attempted to make the “zap” sequence more melodic while Lucy introduced a “space” synth, which created a soft background for the remainder of the mix to sit on. We stripped back the mix to just these synths and minimal percussion, which created a sense of space and a shimmering texture.

(19:50 - 20:45) We had reached the latter stages of the performance at this point and we were beginning to strip a lot of the sound back. Using comments within the code, we decided to finish the performance with something closer to what one might experience at an Algorave. Consequently, I added a deep, but simple, bass line using a player object called `b1` and Lucy began to build up a distorted and gritty rhythm in the background.

(20:45 - 21:40) Laurie created a player object, `z5`, and began to craft a pulse wave sequence that utilised the pitch from the bass synth by using the syntax `b1.degree + (0, var([2,4,6]))` to set the pitch data. It played the same pitch as the bass but also added a second note that would complete a third, a fifth, and a seventh above it for a bar each in turn and sounded great when it came in. This is a very similar musical relationship to the one Lucy made by using the `follow` method and adding a TimeVar to the player object (shown in Figure 6.2) but was created using a player-key instead. Pitch-based relationships are an integral part of our performances and using a player-key in this instance meant that Laurie didn’t have to worry about matching the root note of his chord as it would always compliment the bass.

(21:40 - 22:45) At the same time, Lucy had been working on her own line of code that would

introduce a saw-wave based synth called “star”. She used the pitch from Laurie’s pulse-wave sequence, `z5` but halved the value by using the code, `z5.degree / 2`. Had the degree of `z5` been odd at any point, then the values derived from dividing them by two would have ended with `.5`, causing FoxDot to play a note a semitone higher, i.e. an accidental. However, this did not happen as `z5` was referencing the pitch from `b1`, which used only even numbers.

(22:45 - 25:10) Laurie got disconnected again and we get several error messages appearing in Troop’s console relating to this. Lucy and I continued but we began to experience audio jitter and some samples going out of sync. Laurie managed to log back in but, once again, was assigned a new font colour and it clashes with my own, which may have been very confusing for the audience. As we approached the end of the performance we thought it might be a good time to embrace the crash that seemed inevitable at that point. I added an extremely noisy synth and we started to push SuperCollider to its limit, adding effect after effect to reproduce the glitched artefacts from before. In the process we nearly crashed Troop but it hung on to let us finish and ride out the distorted and glitched sounds.

A.3 International Conference on Live Interfaces, Porto - 14/06/18

Video recording: ch5_3-ICLI-14_06_18.mp4

(0:00 - 02:25) The performance started with each of the performers writing a separate line of code to generate a musical sequence. I chose an ambient drone, Lucy a percussive resonance synth called “space”, and Laurie was working with percussive samples. The drone slowly became a shimmering tone while the “space” sounds were sporadic but haunting and playing a simple sequence on repeat. The percussive samples were being played back at a reduced playback speed and provided a crunching backdrop. After a few minutes the first rotation occurred. I edited the “space” sequence to slide its pitch up and down and Laurie updated the drone. Lucy added a call to “stutter” the percussive samples every 7 beats and they began to appear more frequently, and rhythmically, within the overall mix.

(04:20 - 05:10) We continued to rotate through each others’ code making changes but it wasn’t until the fourth minute that any synth was changed. Lucy switched the drone synth to a pulse-wave based synth, “prophet”, which created a more dynamic backdrop to the rest of the music. The synth sounds were then chopped up, creating an echo effect with irregular and disorienting pulses. I introduced a new, softer drone to thicken the sound. The pitch of the echoing pulse synth was derived from the “space” sequence but modified the value every other note. Although connected, the two layers seemed a world apart.

(05:15 - 06:15) Suddenly there was a large distorted sound caused by Laurie applying a comb delay effect to the percussive samples that disappeared soon after. The frequency of note onsets began to increase and the sound started to become dense. However, within a few moments the number of sound events was reduced and the overall soundscape was taken to a softer and calmer place. This was helped by replacing the “space” synth with a “marimba”, further dampening the tone. By this point the contributions made by each performer were scattered throughout the text buffer and I could no longer keep up with who has done what and the identities of the different sounds were lost. The font colour had begun to merge toward a single colour, further indicating this loss of the individual among the collective.

(08:30 - 10:10) Several percussive sounds were introduced, some bright and some distorted. The rhythms battled against each other and, with that, became more and more dense. We suddenly

found ourselves shifted within the mix as the marimba was replaced by a distorted and alien sound accompanied by a synthesised soprano. We have managed to produce bright percussive tones and low bass notes that could be straight out of a grime track. By this point the colours had nearly completely merged and the individuality is lost; it became very difficult to identify the changes made to the audio by the separate performers.

(10:45 - 15:40) The shift in soundscape forced each performer to develop their musical ideas to accompany this change. A bass line was introduced that followed the pitch from the soprano and resulted in a ghostly and haunting sound. Shortly after its introduction, the bass became much more staccato and harsh, emphasising the foreboding sound that had emerged. For a few minutes the musical ideas at play were explored further by each performer. The soprano sound was then transformed into a more staccato sequence itself using the “chop” effect, perhaps drawing inspiration from the shortened bass notes. The density of the crunchy percussive rhythm was also reduced, seemingly following suit. The bass notes became even shorter and we were left with large amount of empty space in the texture. One by one the sounds were removed but the reverb effect being applied to the remaining sounds created a thicker and warmer texture. The tempo was then increased and just as it felt the piece might ramp up to a crescendo, the performance was over and the sounds faded away.

A.4 Rehearsal session, various locations - 26/04/17

Video recording: ch6_1a-Rehearsal-26_04_17.mpg

(02:10 - 03:15) Laurie started by creating a simple sequence with a bell sound and Lucy used the `follow` method to connect her own “dirt” synth to it. She tried out a few different values to add to her player object; first she added 3, then a triad to create a chord using Laurie’s bell as its tonic, then using a `FoxDot TimeVar` (shown in Figure 6.2) to iterate over the first, third, and fifth note of the chord for two beats at a time. At this point the durations for both the bell and “dirt” synth were the same, which did not make for an interesting interaction but Laurie then uses the `PDur` function to utilise a Euclidean rhythm and create a contrast between the two layers.

(03:15 - 09:30) Lucy set the note onset delay to half a beat and created an offbeat groove within the mix. She didn’t have to worry about what value she uses for pitch as it is always connected to that of the bell sequence and one will always compliment the other regardless of the duration used. We then explored textures for several minutes by adding and manipulating effects on the existing player objects.

(09:30 - 10:20) I changed the bell to a “blip” synth and use the `every` method to “rotate” the melody every 4 beats. This moves every element in the list containing pitch information over by 1 and the first element then becomes the last. Again, even though the pitch data was being altered, the audio generated by the “dirt” synth was still determined by its connection to the other player object though the `follow` method regardless.

(13:25 - 15:00) The jam had become dense and noisy and it felt like stripping away most elements would be the best thing to do. Lucy had introduced a noise-based sequence that was mostly filling out the background of the mix but just after minute 14 she decided to use the `follow` method again to connect it with the fluttering melody created by the “blip” synth. The bursts of noise are not what many would consider “traditionally musical” but the variation between the lower and higher registers created through the connection to the “blip” synth created a surprisingly complimenting combination of textures.

(15:00 - 24:16) Lucy tried out a few different synths and settled on the “dab” synth, which generated more conventional chords to accompany the the “blip”. The jam continued for several minutes but, unfortunately, we experienced a few errors and, from minute 18 onward, we spent most of the time trying to find the source of the issue. The session ended with us all experimenting

with the available layers before being disconnected by a network error.

A.5 Rehearsal session, various locations - 06/06/17

Video recording: ch6_1b-Rehearsal-06_06_17.mpg

(00:00 - 01:30) This session only consisted of myself and Laurie and started with each of us defining a melodic sequence. I assigned a “karp” synth to **r1** and created a simple melody that “rotated” every 4 beats. Laurie instantiated a player object, **z1**, which utilised a viola synth and set it to follow the pitch of **r1** and added the values [2, 5, 7]. This created a relationship between **r1** and **z1** but it was difficult to hear any musical connection between the two sequences. This was due to **z1** deriving its pitch from **r1** but having much longer note lengths. As soon as the accompanying pitch was played by **z1**, the pitch had changed in **r1** and no longer complimented it.

(01:55 - 02:10) Laurie decreased the duration of **z1** and I increased its sustain and lowered it by an octave. At the time I couldn’t put my finger on what was wrong but I was trying to remedy the situation by giving the viola sequence more prominence. It felt like the “karp” sequence should have been deriving its pitch from the viola but it was the other round. It created a strange sense of disconnection between the melodies but without causing any real dissonance.

(02:10 - 04:00) I tried to develop the viola sequence so that it acted as the accompanying background to the brighter melody being played by **r1**; I doubled the length of **z1**’s duration and sustain and also added a fifth on top of the pitch values to strengthen the sound. This did go some way in making the mix feel more complete and the connection between the two layers felt stronger. There is some discussion between Laurie and myself about syntax for a few moments before we continue.

(04:00 - 05:10) I again increased the duration and sustain of the viola and the sounds begin to overlap. The strange interconnection between **r1** and **v1** seemed to compliment the ghostly textures that were being developed. Although not implemented well initially, Laurie and I pursued the music that emerged having used the `follow` method and developed an interesting soundscape with it. Just as it had done previously (see Section 5.3.2) the embracing of an error led us into a great improvisation session and made for exciting and unpredictable outcomes.

(05:10 - 51:15) The `follow` method was not used again for the remainder of the session but it provided a useful platform for beginning the rehearsal and allowed us to easily work on a shared piece of musical material together.

A.6 Together In Music conference, York - 14/04/18

Video recording: `ch6.2-Together_in_Music-14_04_18.mpg`

(00:00 - 01:35) The performance started with each of us working on separate lines of code but with very similar tonality, revolving around the root note of the scale. Laurie began by building up some subtle background chords using the “keys” synth and assigning it to the player object, `z1`. I introduced a soft-yet-noisy drone synth called “klank” using a player object, `k1` and added a varying low-pass filter effect to it to bring it in and out of the foreground. Lucy created a player object called `l1` and used the “space” synth to fill out the higher frequencies of the mix, but still keeping consistent within the overall dark textures of the sound. At 1:27 Lucy updated Laurie’s “keys” sequence to reference the low-pass filter values used in `k1`, using `lpf = k1.lpf`, and tied the two layers together within the mix.

(01:35 - 02:10) At 1:36 you can hear me say “ooh yeah” as I was pleasantly surprised as Laurie began to move his chords a way from the tonic. I reacted by changing the the simple TimeVar I was using in `k1` to `z1.degree[0]` to reference the root note of the chords and further emphasise the new tonal direction of the performance.

(02:10 - 04:00) Lucy seemingly wanted to connect all the active layers together and replaced the notes defined in `z1` with `l1.degree` such that both `k1` and `z1` were deriving their pitch values from `l1`. This meant that `z1` was now using single values as well as chords and that `k1` was referencing its pitch using `z1.degree[0]`. Ordinarily in Python, using this syntax to access a single number would raise an error but the player-key method for `__getitem__` (as shown in Figure 6.7b) checks if the value accessed is a group or not and returns the value if not.

(04:00 - 06:20) Lucy changed the “chop” effect applied to the “klank” synth to use the duration value from `z1` by using `chop = z1.dur`. This created an interesting polyrhythmic effect, as the chopped up elements of the drone overlapped each other with varying degrees. In a similar fashion, Laurie changed the amplitude of `z1` (the “keys” sequence) to follow the values used in the “echo” effect applied to a percussive sequence, `p3`, using `amp = p3.echo`. The “echo” would be value randomly selected from 0, 0.25, 0.5, or 0.75 and the sound of the “keys” synth were dropping in out of the mix.

(06:20 - 09:25) The next few minutes saw us move away from melody and texture and into a more percussive section that explored more rhythmic effects. There was even more experimentation with the player-keys during this period. Laurie applied a “cut” effect, which shortens the duration of

a sample's playback, to Lucy's percussive sequence with `p3.dur / 2` as its input. This created short staccato bursts of the various samples Lucy was using, including a female voice. Up until then the sequence had been quite dominating but this change helped even out its presence against the other punchy percussive layers being developed elsewhere, especially `p3` as the density of that layer's rhythm would vary greatly over time. I then introduced a soft synth called "ambi" into the mix as a backdrop for the dense percussive rhythms to sit on top of.

(09:25 - 11:30) Lucy changed the duration of the "ambi" synth from 8 beats to the value of `p1.dur` so that it would follow the duration of one of the main percussive sequences and also applied a "chop" effect to it. It created a minimalistic alien melody that prompted us to remove some of the density from the performance and focus on a more sparse and sporadic percussive section.

(11:30 - 13:10) The sparsity of events did not last long as Laurie introduced a player object, `z4`, and created a rising melody sequence using the "donk" synth, which utilised Euclidean rhythms commonly found in dance music. A wooden snare-like sound was being triggered on the second beat of each bar, which gave the overall rhythm a regularity that had not been present until this point. Lucy then added the pulse-wave based synth, called "prophet" on top of this, and I brought in a distorted bass using the player object `12`.

(13:10 - 16:15) Lucy updated the "prophet" sequence to use `z4.degree` as its pitch value, taking it from the fast-paced "donk" synth that Laurie introduced earlier. The duration of the "prophet" sequence was 6 beats per note whereas the "donk" sequence used a series of durations all below 1 beat per note. This created a very similar effect to one that emerged in Section 6.2.2 (rehearsal session B) when the player object that was "following" another also had a longer duration. The longer notes produced by the "prophet" synth seemingly held a snapshot of `z4`'s constantly moving melody and seamlessly merged it into the background. At 15:35 Lucy doubled the value by using `z4.degree * 2`, which moved the "prophet" sequence exponentially into the higher registers. Pitch relationships are generally linear and doubling the frequency of the notes could easily have been achieved by moving the sequence up an octave, but the transformation not only moves the pitch into a higher register, but also increases the range of pitches used by the "prophet" sequence, making it more interesting to the listener.

A.7 Algo-Rhythms, Rotterdam, 28/04/2019

Video recording: `ch6_3-Algo_Rhythms-28_04_19.mpg`

(04:10 - 05:10) The soundscape contained several interesting rhythmic and timbral features at this point but was becoming very dense and seemed to be reaching a crescendo, which was emphasised by the rising frequencies of the “prophet” synth. I implemented one of the new features of FoxDot developed during this phase by setting the pitch of the “prophet” synth to `r1.pitch.accompany()`. This created a relationship between the soaring “prophet” notes and the simple harmony being played by the “swell” synth. Interestingly the “prophet” layer accompanied the two notes of the harmony with two different notes, which added to intensifying crescendo.

(05:10 - 06:05) Innocent accidentally left the frequency cutoff for the “prophet” synth at 16, which all but silenced the layer, but was the catalyst for a change in the music. Several changes to parameters throughout the code which slowly started to add more space to the overall sound. Innocent added a “pluck” layer with a duration of 1/3 and also used `r1.pitch.accompany()` as its pitch input, which created a tonal relationship to other layers in the music, but also created a sharp rhythmic contrast as it punched through the long sustained notes currently at play.

(06:05 - 08:00) Lucy tentatively began to type `r1.stop` before deleting it, as she saw there were two layers using it as the source for pitch information. However, I did feel that the music needed to be stripped back and decided to stop it myself in an attempt to move on to a new section in the music. I replaced the `r1.pitch.accompany()` in the “prophet” synth layer with a simple melody and began to work on the sequence. Shortly after I did the same for the “pluck” sequence. Innocent stopped the “spark” sequence and suddenly the soundscape feature almost no bass frequencies at all and felt very empty.

(08:00 - 10:20) The sparse atmosphere was short lived as Laurie introduced a growling “saw-bass” synth to a layer called `zx` to fill out the mix. This was soon accompanied by a bright and choppy “star” synth layer created by Lucy that used `i2.degree * 2` (double the pitch value of the “prophet” synth layer) as its pitch input. To compliment the more fleshed out soundscape, I increased the sustain on the repeated “pluck” layer and added a repeated to call to the “offadd” method.

(10:20 - 11:40) Innocent introduced a new layer, called `i4`, using the “nylon” synth and created a relationship with the bass line Laurie had developed by using `zx.pitch.accompany()`. Laurie

responded to this focus on the bass by adding the `zx.degree` value to the melody used by the “pluck” synth layer. This added a nice level of variation to the melody, which had a polyrhythmic feel to it having been combined with the “offadd” method.

(11:40 - 13:00) I decided to try out the new `versus` method by calling `versus(i3.pitch)` on the `i2` layer. The result was that the “pluck” and “prophet” synths started to take turns to play, combining to create a call-and-response. This also helped create a bit more space in the mix without explicitly removing a layer, and the indeterminacy kept the music feeling organic.

(13:00 - 13:35) I decided to silence every layer apart from those with names starting with “i” to try and emphasise the use of `versus` but the amplitude values used in the `i2` layer meant that it did not always generate sound when its pitch value was greater than its `versus` counterpart, `i3`. This perhaps created too much space in the mix and I added back in the percussive sequence. Even if this did not work the way I had intended, it did create a sharp contrast to the density and timbre of the music.

(13:35 - 14:05) Another interesting outcome of my decision to silence several layers at once was that it affected the pitch relationship created by Innocent in the `i4` layer, which was using `zx.pitch.accompany()`. The `zx` layer had stopped, so `i4` had nothing to accompany and played the same note on repeat until we brought back in the bassline using `zx`. The single repeated note developed a level of tension, which was then released as the accompanying relationship was resumed with the reintroduction of the bass.

(16:00 - 17:20) Lucy’s interface froze and her cursor got stuck in the middle of the line. It wasn’t until Laurie wrote a comment in the code that Innocent and I were aware of the problem but we tried to continue while Lucy examined the issue. Lucy was not able to log back in and it seemed the best course of action was to close everything down and restart.

(17:25 - 18:35) After a few minutes delay while we restarted the Troop server and reconnected, we resumed our performance. I started by adding a breathy drone synth, called “klank”, while Lucy created a sequence of robotic sounds and Laurie introduced the “donk” synth with a layer named `zc`.

(19:30 - 22:00) Lucy added a “marimba” synth that used `zc.pitch.accompany()` as its pitch input. The `zc` sequence was using a `linvar` to generate microtonal pitch values that linearly oscillated between the root note and its fifth. I had not foreseen that the `accompany` method

would be combined with these sorts of pitch values and it was interesting to hear the results. The “marimba” synth was also playing microtonal notes, which creates a harsh clashing of tones but complimented the other layers thematically. We hadn’t discussed the use of microtonal work prior to the performance, and it was surprising to see it make an appearance, and the use of the `accompany` method helped keep the theme consistent across multiple layers easily, while adding variety to the process.

(22:00 - 23:55) Lucy changed the tempo to 130 beats per minute, then 140 shortly after in an effort to ramp up to a conclusion. Following that thread, she also replaced the delicate notes of the “marimba” synth with a “varsaw”, which drastically altered the overall soundscape to something much noisier and disruptive. The microtonal notes felt too strong at that point so I decided to alter the `zc` layer to only play a microtonal value every third note and alternate between the root and fifth for the other notes. Meanwhile, Innocent created a new line using the “prophet” synth that did use `zc.pitch.accompany()` as a text input, but with a longer duration of 8 beats per note. Lucy also added a pitch-shifted and distorted snare drum on repeat, which added to the urgency of the outro.

(24:45 - 25:55) Laurie writes “strip back?” as a comment, suggesting that we start to remove elements from the mix to fade out the music. Interestingly, Innocent adds a heavy kick drum with a low-pass filter applied, which starts to carry the beat. While Lucy and Laurie start to remove some layers, I decided to spread the kick drum out a bit as not to completely undo Innocent’s code, but to try and fit it in with the “stripping back” process that was happening at the time. Lucy and Laurie then removed all synth-based elements and we were left with just two percussive sequences. Lucy then cleared the clock and brought the performance to a well-executed close.

A.8 Rehearsal session, Sheffield - 09/12/18

Video recording: ch7_1-CodeBank_Rehearsal-09_12_18.mpg

(00:00 - 02:50) The session began with someone introducing a simple hi-hat rhythm that stuttered itself every few beats. After a minute a low, droning tone was introduced at 01:20 and was shortly followed by short and sporadic notes from the “keys” synth. At 01:53 I took my headphones off briefly to listen to the mix before deciding to pull the codelet containing the “keys” synth from the public repository. It isn’t until 02:30 that an audible change to the soundscape (an increase in pitch of the drone) was heard. At almost the exact same time, both a “space” and “donk” synth were introduced. It felt like the result of everyone’s musical ideas were realised simultaneously and seemed to create a sense of unity about the music.

(02:50 - 03:30) Just after the 3 minute mark I pushed my updated codelet into the public domain, which introduced more bass frequencies into the mix and created a thicker and warmer texture. The changing chord sequence also helped add some variety to what had become quite a repetitive and monotonous sound.

(03:30 - 05:10) Lucy introduced a heavy kick drum sequence with a 5/8 over 8/8 polymetric rhythm, which added momentum to the music but without an overly obvious beat. At around 04:30 Laurie added a low-pass filter to the kick drum sequence to remove the treble from its attack and smooth out its sound to blend better with the mood of the rest of the music. He had spent much of the session up to this point with only one headphone on, listening to the mix and obviously felt that the EQ of the drums needed adjusting.

(05:10 - 06:20) At this point, a stuttering effect was added to the kick drum layer; echoing the build up of the hi-hats that started off the practice. The “space” synth was then lifted into a higher octave, creating a nice contrast within the mix, especially against the low frequencies of the kick drum and “keys” sequence.

(06:20 - 08:15) Everything seemed to be coming together nicely and each different element added by the group complimented the dark and atmospheric mood of the music. We continued to develop our ideas further for several minutes, creating a rich and dense sound that was starting to build up a lot of tension.

(08:15 - 09:15) An arpeggiated synth was added and it immediately cut through the mix, sig-

nalling an impending musical change. Over the course of the following minute we began to strip out sounds from the lower frequencies and started working on brighter and more melodic sequences, while still keeping the driving kick drum and hi-hat combination.

(09:15 - 10:30) We began to strip the layers back further still by replacing the hi-hats with soft burts of noise and clicks. The distorted bassline that had been pulsating in the background now pulsed less frequently, creating a soft 3/4 polyrhythmic rhythm against the even brighter synth notes in the foreground.

(10:30 - 12:30) The scale was changed from major to minor and a percussive note with a quick rhythm was immediately introduced, giving impetus to the new musical direction. No further change occurred until 11:10 when the tempo was increased, adding an even greater sense of urgency to the music.

(12:30 - 13:45) A snare-drum rhythm reminiscent of an army marching band was introduced at the end of minute 13, but was changed to hi-hat at 13:27. At 13:40 the kick drum sequence was put through a high-pass filter, which created a real sense of tension as if the music wasn't quite complete without the bass frequencies of the drum.

(13:45 - 15:25) A descending and distorted melody was added, which took centre stage within the mix immediately. Short flurries of kick drum samples were played before being taken away and returning to that sense of tension. It wasn't until 14:55 that the kick drum returns for good, adding a steady beat, filling out the mix and carrying the momentum of the music.

(15:25 - 17:15) At 15:30 a delicate high-pitched pulse-wave synth was added that sat above the distorted melody. It expanded the tonal range of the music and added a new dimension to the sound. A high-pass filter was applied to the kick drum sequence for the last bar of the eight-bar cycle, adding a small amount of tension before releasing it.

(17:15 - 17:50) The distorted melody was replaced with dissonant synth chords, adding tension and a sense of foreboding to the mood, possibly indicating greater musical changes were to come.

(17:50 - 19:10) The kick drum was changed from four-to-the-floor to a three-over-eight Euclidean rhythm, which removed a lot of the density from the mix and slowed the pace of the music down. At 18:02 an accordion-sounding synth was introduced along with sporadic clap samples, creating a sort of dark electro-sea shanty vibe, which was definitely not something we had planned on

exploring at any point in the rehearsal.

(19:10 - 20:00) A throbbing bassline was added, which shifted constantly between metric and polyrhythmic rhythms. The accordion sound was stripped back to only play in the last bar of a four bar cycle, but provided an interesting contrast to the lower frequencies of the bass.

(20:00 - 20:40) The four-to-the-floor kick drum was re-introduced along with sporadic dubstep snare samples, which created a steady, yet off-kilter beat. A barrage of cowbell and woodblock samples were added to the mix, but only for a moment as Lucy pushes a codelet that contains an error and stops all the sound.

(20:40 - 23:55) Lucy asked “Why is it pink?” referring the colour of the codelet she had just pushed the public repository. I hadn’t properly explained this feature of the system yet and, after clarifying what she had asked, I reply “Oh, that means there’s an error”. There was no error raised when running the code in her private workspace, but the codelet was coloured pink when pushed and also caused the audio to stop completely. The next minutes consisted of debugging and discussion and we realised there was a Type Error¹ being raised because the Lucy had changed the player object’s input to a string, but it was being referenced as a player-key by another codelet, which required a number.

(23:55 - 24:50) We resumed the jam and Innocent introduced some new samples including a male voice shouting “huh”, much to Lucy’s delight. The kick drum was slowly stripped back until it only played on the first and last beat of the bar, creating a lot of space in the mix. Further space was made when the cowbell rhythm was removed shortly after.

(24:50 - 26:15) The array of samples that Innocent had brought in were sped up and played quicker, creating a funky and minimalistic beat.

(26:15 - 26:41) A simple voice-like synth chord was added that alternated between two notes but it added an element of seriousness to music, which had become quite playful up to this point. It was starting to feel like we were building up to a slick IDM section when the battery in the camera ran out and cut the recording tantalisingly short.

¹Ironic.

A.9 TOPLAP End of Cycle Party, Access Space, Sheffield - 19/12/18

Video recording: ch7_2-TOPLAP_End_of_Cycle_Party-19_12_18.mpg

(00:00 - 00:15) We synchronised to the CodeBank server using the ‘clock nudge window’ and a single clap sound to make sure what we heard in our headphones was identical to what we heard through the PA system. Once we were all synchronised we began to play.

(00:15 - 01:40) Since the performance was taking place so close to Christmas we decided to start with something one might describe as “festive”, which involved sleigh bell and tambourine samples, and the occasional woodblock. We started to introduce a few synths that played a single repeating chord with an offbeat rhythm and a bright melody in the upper octaves that definitely fit the theme of “festive”.

(01:40 - 03:30) A simple drum beat with a low-pass filter applied was then added to carry the music and give the audience something to nod along to. The bright melody soon began to run up and down the scale and a mix of hi-hats, maracas, and castanets and were introduced on top of the drums, adding to the groove.

(03:30 - 05:30) A soft, but definite, snare drum came in on every other beat and helped keep up the momentum but the overall sound started to become murky, overly dense with percussion, and repetitive.

(05:30 - 07:30) The tempo was changed to 140 bpm and the bright synth sounds were removed to leave only the layers of percussive samples and a single synth tone. This helped alleviate some of the issues with the music and ended up creating quite a nice groove.

(07:30 - 08:50) The droning synth changed into an offbeat and staccato stab in the lower octaves and fuelled the groove we had developed. The kick drum layer was removed from the mix, creating some tension that was swiftly released following the reintroduction of the kick drum and also the addition of multiple snare samples.

(08:50 - 10:25) More synth presence within the mix created a darker atmosphere to the music and the pace was slowed down by removing every other kick drum sample. The build up in suspense

was being very well coordinated.

(10:25 - 11:35) A deep and distorted bass synth seemed to suddenly scream in the background and the performance was given a jolt of energy. The slower kick drum stopped things from escalating further but it was definitely indicative of a more high-octane musical direction.

(11:35 - 11:55) In contrast to the dark and gritty bass, a soaring “soprano” synth was introduced and the kick drum was changed back to four-to-the-floor. Things came together perfectly here and there were definitely some bodies moving in the crowd.

(11:55 - 14:00) A formant filter was applied to the drum samples, which removed most of the bass frequencies and created a level of tension. This tension was released as the filter was removed at 12:08, bringing back the high-energy electronic dance music. In true EDM fashion, we added a “drop” at 12:28 to try and surprise the audience and give them a break from the repetition.

(14:00 - 15:00) At 14:07 a distorted synth with sharp attack was “solo’d”, which left it as the only layer playing. The sequence’s varied rhythm was of great amusement to Laurie who bent over laughing to himself. It’s always fun when something surprises you during performance! Each layer was gradually added back in and by 14:50 we were back with a strong and steady beat.

(15:00 - 16:25) A single note with a reversed envelope was added on repeat before its duration was stretched to 8 beats. It seemed that some performers had become out of sync with the server and were not able to listen to the audio in private workspace properly, causing them to introduce elements into the public repository to listen to them instead.

(16:25 - 18:10) The tempo was slowed down to 70 bpm for a few bars to create the effect of a “break down” but was quickly changed back to 140 bpm to give the performance a boost of energy. However, the bassline was slower and the level of distortion applied to it increased with every beat, resetting after 8 beats. This coincided with a whirring tone that would reach a crescendo at the same point.

(18:10 - 18:40) At this point Lucy turned to me to ask about moving on to develop a new musical idea. Instead of using the chat box in CodeBank, the message was passed down to Laurie, who then passed it down to Innocent, which took nearly 25 seconds.

(18:40 - 19:25) A barrage of distorted and glitchy samples were added, accompanied by a range

of snare drums that appeared sporadically before returning to a regular beat at 19:05. Laurie and Lucy shared a joke, causing them to laugh. Having co-performers in a good mood on stage always helps alleviate any nerves and makes for a more enjoyable performance.

(19:25 - 20:50) At 19:50 a lot of the main drum layers were removed from the mix, although some of the higher frequency percussion remained. The kick drum was reintroduced along with a seemingly random selection of other samples. Several sets of headphones were not being worn and many more smaller and incremental changes being made to the public code, which suggested that we were not making as much use of the private workspace as earlier in the performance.

(20:50 - 22:15) A single high-pitched note was introduced and the bassline was altered to slide up in frequency over several bars, creating an element of suspense. At 21:50 the single note was also updated to rise in pitch in the same style as the distorted bass, further adding to the build up in tension.

(21:15 - 22:45) I added a clap sample on every downbeat before doubling and quadrupling its speed shortly after, creating a continuous wave of noise. I was not testing these changes in the private workspace as I felt that their addition (and removal at 22:45) was time-critical.

(22:45 - 23:50) The music continued to loop for another minute, seemingly building to a crescendo that never came as the music was stopped midway through the last bar of an 8 bar cycle. A rising synth line continued to slide higher in frequency indicating a resolution but instead faded away and the performance was over.

A.10 Late at the Library: Algorave, London - 05/04/19

Video recording: `ch7_3-BritishLibrary_Algorave-05_04_19.mpg`

(00:00 - 01:30) We decided to start the performance slowly and build up the textures and melodies before introducing too many percussive elements. I introduced a soft noise-based drone, which was soon accompanied with bright synth chords and a simple, repeated “crunchy” rhythm.

(01:30 - 02:10) A harmony was added to the drone and, after listening to output in my headphones, I added a faster rhythm made from percussive samples being played between 5 and 10 times faster than their usual rate.

(02:10 - 03:10) A soft kick drum was then introduced together with a ‘pluck’ sound, which gave the music a little more impetus and energy. The amount of noise in the drone momentarily soared above the mix before fading into the background as some syncopation was added drum beat. All three performers put their headphones on as we started to work in the private workspaces and looked to introduce a change to the music.

(03:10 - 04:20) A soft bass synth was added and shifted the tonality of the sound slightly. This change was accompanied by the increase in density of the “crunchy” percussive layer. It was no surprise This came just after the kick drum sequence was changed in a similar manner and this change complimented it well.

(04:20 - 05:30) A synth with a sharp attack playing a major seventh chord was added using the Euclidean rhythm 5 over 16, which dominated the mix for what felt like a long time. The samples in the kick drum sequence were played back at 4 times their normal speed for the last few beats of a 4 bar cycle, which started to add some tension to the music.

(05:30 - 06:35) I turned to the others to say “I’m gonna add a heavy kick in now” as I felt that, with tension in the music rising, it was the right time. I decided not to write this message in the chat window as I didn’t want to let the audience know about the upcoming change. While working on this, the synth playing the repeated major seventh chord was updated to follow the pitch of the bassline, using the player-key syntax, which created quite a large shift, and variety, in tonality.

(06:35 - 08:05) The four-to-the-floor kick drum came in next and added a lot of energy and drive to the music. At the 7 minute mark, a simple percussive layer was added, which seemed to

replace the “crunchy” sequence that had been in the background previously, consisting of samples of finger-snaps, hi-hats, and snare drums. In the same instance, a shifting low-pass filter (with a varying degree of resonance) was applied to the major seventh chords, which not only added some timbral variety, but also balanced the overall mix. The simple percussive layer was emphasised by replacing soft snare drum samples with heavier ones that might appear in a dub-step or drum and bass track. In the context of the light mood of the music we were playing it had a strange effect; seemingly slowing down the piece by spreading the beat emphasis over a great period of time.

(08:05 - 09:25) The bassline was removed from the mix, which also killed any player-key relationship that existed with it, such as the major seventh chords. This simple change removed any sort of change in the music regarding pitch and signalled some sort of change was coming. The heavy kick drum sequence was also stopped, further heightening the suspense. A fast hi-hat sample was added just as the chord stabs were also removed, leaving just the bright chords stabs that were introduced at the start of the performance and some light percussion. There was definitely an impending change in the music.

(09:25 - 10:50) After a short discussion, the scale was changed to the mixolydian and a kick drum was added at the start of each bar. At 09:55 The bright synth chord stabs were changed to a simple repeated riff using first and seventh note of the scale and moved down one octave. Combined with the sawtooth texture of the synth, this shifted the atmosphere of the music to a darker mood.

(10:50 - 12:20) The density of percussive samples was building up and a male voice shouting “aye” could be heard. I added a bright sitar counter-melody to contrast the dark synth riff, which seemed to take inspiration from classical Indian music. There was a bit of a clash between this and the newly added bassline and the music ended up with a “muddy” quality. This was fixed soon after, with the bass playing the root chord notes for the first, seventh, and fourth notes of the scale.

(12:20 - 13:45) The dark sawtooth riff was made much brighter by moving it up two octaves and chopping the sound up to create a rippling effect to the notes. A kick drum was introduced at the start of every bar, whose duration was slowly decreased over several bars until it reached a four-to-the-floor rhythm. I applied a high-pass filter to the overall mix for the last bar of the 8-bar cycle so that, at 13:42, all of the bass was removed, developing a moment of tension, before being added back in at the start of the next bar, creating a “drop” effect.

(14:30 - 15:40) A deep distortion effect was added to the bass notes, with the intention of adding a psychedelic feel to the music. Shortly after, all percussive sections were removed, leaving only the sitar and bright synth melodies playing.

(15:40 - 16:30) Starting with an off-beat shaker sample, percussion was slowly added back into the mix. This was followed by a soft kick drum and the snare-drum pattern from minute 7, but played at a faster playback speed to create the effect of a woodblock sound. After these, the psychedelic bass notes were introduced, followed by the four-to-the-floor kick drum, which got a good reaction from the crowd.

(16:30 - 18:20) We changed the scale to minor, which transitioned quite awkwardly. The bass and the bright synth riffs were using notes that appeared in both mixolydian and minor scales so only the pitch of sitar melody changed in a meaningful way. Eventually the change propagated across to the bassline as it played some minor notes, but it did not feel very well co-ordinated at the time. We removed several layers of the mix such that we were only left with the distorted bass and minimal percussion and the occasional male voice sample shouting “aye!”.

(18:20 - 20:45) The voice sample reminded me that there are also some beat-boxing samples in FoxDot and I decided to create a simple rhythm using these and the `amen` function that replicates the rhythm of the “amen break”. The mix had started to feel a little ‘close’ since the level of distortion and reverb on the bass had been reduced and, to combat this, the “space” synth was introduced. It added a great contrast to the short staccato beat-box rhythm and complemented the the bass well. Innocent used the new `accompany` feature that had been added to FoxDot to tie together the “space” synth with the bass through pitch.

(22:15 - 23:30) The beat-boxing samples were re-introduced and accompanied by a mix of male and female vocal samples. An overdrive effect was added to the bass, intensifying the dark and brooding atmosphere that was being developed. At 23:00 you can hear a formant filter being applied to the bass, which, when combined with the overdrive effect, generates a gnarly sounding electro-bass that brings in more overtones as the input values for the filter increase.

(23:30 - 24:35) The beat-box samples are once more removed from the mix, leaving the off-beat shaker sample as the only percussion. The bass’ pitch is alternated every other note and is soon accompanied by a short off-beat synth and ride cymbal. The whole sound seemed to rumble, creating the feeling of suspense, which was intensified as the formant filter continued to distort the rhythmic bassline. The suspense was soon dissipated as a heavy hitting kick drum was introduced

at 24:10, which injected the set with a large hit of energy.

(24:35 - 25:35) I added some waveshape distortion to the kick drums to soften their impact but keep the four-to-the-floor rhythm intact. Similarly, the impact of the distorted bass was also reduced by shortening the sustain, removing any overlapping bass sounds. The distortion on the kick-drum was removed and we returned to our dark and dense loop.

(25:35 - 26:10) The offbeat ride cymbal was replaced by a much more staccato hi-hat sample and the kick drum sequence was “stuttered” every few beats. There was a conversation between the three of us and we agreed to make a more drastic change to the music.

(26:10 - 27:25) The kick drum was slowed down drastically and the distorted bass removed. The breathy, sustained synth and sporadic offbeat stabs were left fluttering in the higher registers against a contrasting use of the “donk” synth that utilised much lower frequencies to create percussive sounds. At 27:05, a bright and echoing note was added, using a synth called “pasha”, as the kick drum sequence was stopped completely.

(27:25 - 28:00) We continued to strip the layers back to leave a few synths playing sporadic and spaced out notes, accompanied only by a hi-hat sample played on the second and fourth beat of the bar. A swelling synth would occasionally push through the foreground only to fade away and leave the echoes of the “pasha” and “donk” synths scattered polyrhythmically throughout the foreground.

(28:00 - 28:40) A repeated bell synth was introduced by Innocent, who was no longer using headphones. Even though it played just a single note at a 1/4 beat duration, it combined well with the “donk” layer to create a nicely syncopated rhythm. At 28:27 you can hear a screaming burst of noise, which was created by Laurie when he changed the playback rate of the crash cymbal that had been triggering every 8 bars. This seemed to symbolise the next movement of the set.

(28:40 - 29:25) The “pasha” synth was updated to play a short and repeated melody instead of random notes at random times. There was another brief conversation between the performers then the kick drum was re-introduced slowly. The pitch of the bell was changed to be dependant on that of the “pasha” sequence, which added tonal variety to the layer as well as its rhythm.

(29:25 - 30:40) Again, a four-to-the-floor kick drum was re-used but within the context of the current soundscape it felt very different; a brighter timbre and notes in higher registers created

an almost 'oriental' atmosphere with plenty of space in the mix. Even though the tempo was the same as it had been in previous sections, it didn't feel as intense because of this. The player-key relationships being used by Laurie and Innocent created warm sounding harmonies that didn't constantly repeat themselves, and kept the melodies and counter-melodies interesting to the listener.

(30:40 - 31:40) A distorted bass line was added to flesh out the lower frequencies of the mix, which had been focused primarily on the higher end of the spectrum. An out of tune note created by Laurie using a `linvar` as pitch would sound in the last few beats of a four bar cycle, creating a brief dissonance and providing a sharp contrast to the harmonies that were currently present in the mix.

(31:40 - 32:10) A snare roll was added for the last two bars of every four bar cycle to build some suspense for the ending. This occurred four times before we scheduled a synchronised clock stop at the end of the final snare roll and finished the set.

A.11 Rehearsal session, Sheffield - 07/05/19

Video recording: ch8.1-Rehearsal-07.05.19.mov

(00:00 - 00:35) We introduced percussion sequences in both TidalCycles and FoxDot to make sure we were in sync and then updated them to add some syncopation and groove.

(00:35 - 00:55) Some code was already written in both the FoxDot and TidalCycles text buffers that define short melodic structures. We ran the code, which triggered the simple SynthDef that was defined in the SuperCollider text buffer. The synth was based on a triangle wave multiplied by a pulse wave, which gave it a slightly grainy effect.

(00:55 - 01:55) The amplitude of the triangle wave was increased, as well as the frequency of the pulse wave, which intensified the grainy texture of the synth. The pulse wave was then removed altogether, leaving a very soft timbre sound behind.

(01:55 - 03:20) I replaced the triangle wave with a sawtooth wave, which created bright and strong tones. At 02:50 I lowered the frequency of the synth and gave the whole soundscape a new tonality, before returning to the previous value. Laurie increased the sustain for the FoxDot sequences, which gave the synth more emphasis and time to breath within the mix.

(03:20 - 04:00) Laurie then doubled the frequency, moving the pitch up an octave. The frequency was modulated by a “line” input, which increases or decreases a value over time, and changed the frequency over a period of 16 seconds. Longer notes can be heard sliding their frequency whereas shorter notes are closer to the root frequency of the note they are playing.

(04:00 - 05:05) We both decided to work on the existing musical sequences for a few moments; Laurie using FoxDot and myself using TidalCycles. TidalCycles is a very good language for the algorithmic manipulation of patterns and, through the use of only a few simple functions, I was able to remix the TidalCycles melody. I wanted to make a more drastic change, though, and at 05:00 I set the sequence to play just one note per cycle.

(05:05 - 07:35) I started to develop a running melody over the chromatic scale in TidalCycles while Laurie worked with the synth’s envelope, changing both its levels and duration.

(07:35 - 08:45) I decided to change the saw wave to a variable saw wave, which had a surprisingly

significant impact on the timbre of the sound. Combined with its envelope, the synth was generating “bleep” sounds that might be expected to come from a sine wave oscillator instead of a saw wave. I opened the SuperCollider window to check the input arguments for the variable saw wave and see how I could manipulate it. I changed the “width” argument to a sine wave, which created a much brighter timbre, which, at times, sounded like a cheap imitation of a trumpet sound. Meanwhile, Laurie had been developing the FoxDot code and updating the pitch input, which added a level of variety to the tonality and combined well with the sustained and overlapping notes.

(08:45 - 09:45) The change in tonality prompted larger changes in the music; I started to create a dense and noisy, but simple, sequence using TidalCycles’ “gabba” and “off” samples. I increased the density to the point where the sounds blended together to feel like one single tone, before suddenly undoing these changes and returning to a simple rhythm. All the while, Laurie had been making subtle changes to the synth’s frequency by shifting it several hertz over time.

(09:45 - 11:05) The overall density of the mix had also increased at this point, but this was counteracted by Laurie’s changes to the synth’s envelope, which cut the sound short and put an end to the overlapping notes sitting in the background. I started to change the pitch input for the melody defined in TidalCycles. At 10:50 I divided the frequency input for the synth by 8 to move both FoxDot and TidalCycles sequences down 3 octaves. I then changed the divisor to 4 in place of 8 to move the pitches back up an octave, but still keeping a darker and grittier texture to the sound.

(11:05 - 11:30) I then decided to try and add a choppy effect to the synth by multiplying it by a pulse wave with a short period. However, after the first period, the synth would go silent as SuperCollider would think the note had finished playing and then remove it from memory. This unintentionally resulted in a short period of deep staccato notes. Instead of trying to pursue my original idea, I increased the frequency of the pulse wave to create a densely choppy synth sound that had a ‘robotic’ quality to it, similar to the timbre at the start of the rehearsal.

(11:30 - 12:26) I tried manipulating the frequency input for the pulse wave but could hear no changes to the timbre and, so, had to open SuperCollider several times to check the contents of the console. Not getting console output in the interface is definitely one of the downsides of communicating with an environment using OSC as opposed to directly through subprocesses as Polyglot does with TidalCycles and FoxDot. Laurie, meanwhile, flirts with adding a “chop” effect manually to the synth sequence in FoxDot before removing it and focusing on changing the pitch input instead.

A.12 AlgoMech Festival, DINA Club, Sheffield - 18/05/19

Video recording: ch8.2-AlgoMech_Festival-18.05.19.avi

(00:00 - 02:40) The performance started with a simple drum beat being introduced using Tidal followed shortly by a bass and a siren-like synth note on repeat created in FoxDot. There were some technical issues with connecting Lucy’s laptop to the projector and weren’t able to remedy this for a several minutes.

(02:40 - 03:05) We eventually managed to connect Innocent’s laptop to the projector and were able to focus on the code as opposed to the technical set-up. We started to embellish the percussive elements by introducing high-frequency bursts of noise. At 03:05 you can start to see that the location of each performer’s text cursor begins to blur as Polglot stops refreshing properly. It eventually corrected itself, but was a sign of the technical issues to come.

(03:05 - 05:00) A “pluck” synth was added in the FoxDot text box, which played a single note on repeat. At 04:17 the pitch of this synth starts to alternate between the tonic and the mediant, which creates a contrast against the darker, almost industrial tone of the music.

(05:00 - 06:00) This tonal shift prompted us to pursue a “psychedelic” aesthetic, which is something we had visited before in performance. The octave of the both the “pluck” and bass sequences began to move up and down at seemingly random intervals and a hard kick drum was added using Tidal.

(06:00 - 07:00) The sustain of the “pluck” sequence was then shortened drastically and it virtually disappeared from the mix. The sustain value was set using a `linvar`, which would increase and decrease over time and bring the “pluck” layer to the foreground briefly before fading away. Meanwhile, the Tidal-based drum sequence was being stripped back to leave a sparse beat.

(07:00 - 08:25) A “space” synth was introduced in the FoxDot text box and the bass was given a new set of pitches to follow. The “pluck” synth, connected to the bass using the FoxDot `accompany()` syntax, seamlessly began to “play along” with this new pitch data. I added an offbeat clap and noise sample using Tidal and we started to pick up the momentum of the performance.

(08:25 - 09:00) Innocent turned to me and explained that the editor had become virtually unre-

sponsive to him and was not properly updating every performer's text cursor. I guessed this was likely caused by the amount text in each buffer so I decided to remove as much as I could from the FoxDot text box without disrupting the music. This helped give Innocent some control over text input but the issues with the text cursors continued for some time.

(09:00 - 11:20) The bass line chord sequence was simplified and the use of the minor key combined well with the noisy and off-kilter drum patterns created in Tidal. Innocent suggested that I project my screen as my laptop was more powerful but after some time searching I could not find the necessary adapter and I had to ask Laurie if he could do the projection. Unfortunately there was also no available adapter for him and Innocent had to remain as the projector-facing performer, even though his interface appeared the least responsive.

(11:20 - 12:15) To try and improve Polyglot's responsiveness for Innocent, we stripped back as much code and audio as possible but still Innocent's GUI would go white whenever he tried to interact with it. We decided to continue the performance to the best of our abilities but it meant that Innocent could barely contribute.

(12:15 - 12:40) I began adding a drum sequence in the Tidal text box using the "feel" sample bank. I didn't know what type of samples it contained but I had seen it used in other performances and decided to just play all of the samples in a row and, luckily, it generated a distorted drum loop with a lot of energy.

(12:40 - 14:20) The scale was changed in FoxDot, which caused a large tonal shift in the bass line. I didn't have a lot of experience using Tidal so, using a comment, I asked Lucy to come and join me. Sustained synth notes began to appear in the background of the mix as Lucy began to write some Tidal code. I started experiment with the number of samples played in one of Tidal's cycles, alternating between 8 and 6, which would create a polyrhythm every 4 cycles. Lucy introduce sporadic and stuttered clap samples to add to the frenetic atmosphere that we were building up.

(14:20 - 15:00) The bass' pitch sequence changed and the background synths were given much more prominence within the mix as the percussive sequences seemed to become stripped back, only to return with bursts of frantic claps and snare drum hits.

(15:00 - 16:25) A distortion filter was applied to the bass sequence that started to growl as the amount of distortion increased over several bars then reset. This rough texture seemed to juxtapose the Major-key chord sequence that was being utilised by the bass and the scale was changed once

more to reflect the darker mood. Innocent tried to find a workaround for his unresponsive interface by opening a separate text editor and copying and pasting text into the Polyglot editor, which seemed to give him some success.

(16:25 - 17:40) A regular clap sample was introduced on the third beat of the bar using Tidal, which helped increased momentum of the music. The synths in the higher register played off of the dark and grungy bass line, combining to create harmonies and dissonance and helped develop the ominous atmosphere.

(17:40 - 18:45) The distorted bass was then removed momentarily and a low-pass filter applied to the clap samples to move them into the background. We were left with dissonant and ghostly synth notes on top of a variety of percussion before the bass was re-introduced but in alternating octaves this time. Again Innocent had to open a separate text editor to copy and paste text in.

(18:45 - 20:20) Many of the percussive elements were removed, leaving behind the menacing combination of the distorted bass and synth. My face began to be lit up by my computer screen as I was checking the names of samples in the TidalCycles samples library and I wanted to add a new sample. I chose “wood”, which added a rhythmic sawing of wood sound. I replaced this quickly with a more simple “can” sample. The existing sequences were elaborated on, but still within a very stripped back sonic backdrop.

(20:20 - 22:00) Lucy manipulated some Tidal code to create a glitchy rhythmic using the “can” sample and I opened a SuperCollider SynthDef in the appropriate text box to try and incorporate some sound design elements into the performance. Innocent added some synth stab samples as Lucy continued to elaborate her glitchy, almost mechanical-sounding, layer from within the Tidal buffer.

(22:00 - 23:10) A kick drum was added into with a four-to-the-floor beat to try and give the music some energy. I was trying to edit SuperCollider code but could not hear the results of changes and had to continually open SuperCollider to check the console for feedback on Syntax errors and I spent a good deal of time trying to deal with this problem.

(23:10 - 25:00) Some more percussion-based sequences were added and the TidalCycles patterns were elaborated on by Laurie and Lucy. I continued to try and run SuperCollider code and update a SynthDef but struggled to get it to work as I was not able to identify if there was a syntax error or not. At 24:40 you can clearly hear me say to Laurie “yeah I tried that but it’s not working” as

he makes suggestions on how to fix the issue.

(25:00 - 25:55) I decided to leave the SuperCollider code alone and add a snappy snare drum using FoxDot to try and up the energy levels for the final five minutes of the performance. This was shortly followed by rapid hi-hats as we aimed for a big finish.

(25:55 - 28:00) An electronic bass was added that derived its pitch from the higher-pitched synth sounds. Slowly more filters and effects were applied until the bass became a wobbly and de-tuned alien voice. This was offset by a burst of noise created using Tidal occurring on the offbeat of the last beat of the bar. By minute 29, however, the alien bass sounds were removed, leaving the high-pitched synths as the most prominent feature once more.

(28:00 - 29:10) At 28:30 the majority of the percussion layers in both the FoxDot and TidalCycle text boxes were removed, leaving only the overlapping synths playing and finishing on a melancholic note.