UNIVERSITY OF CALGARY

JunctionBox

A Multi-touch Interaction Mapping Toolkit for Creating Musical Interfaces

by

Lawrence Fyfe

A DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTATIONAL MEDIA DESIGN

CALGARY, ALBERTA

AUGUST, 2015

Abstract

This thesis describes my research into the development of a unit interaction model for multi-touch interactions in a musical context. To create this model of unit interactions, I first determined the most fundamental aspects of multi-touch that offer interaction building blocks that can be combined in a variety of ways, allowing for a high degree of freedom to design and build musical interfaces. This unit interaction model is implemented via JunctionBox, a toolkit for mapping multi-touch input to control of music.

With JunctionBox, composers, musicians, and programmers can build interfaces that combine multi-touch and mapping for use in a wide variety of musical contexts. As a toolkit, JunctionBox features multi-touch input tracking, mapping of input to output via messaging, output for graphical feedback, and flexible networking options. All of these features are designed such that they can be used in any combination, allowing for tremendous creative freedom in building interfaces.

To put JunctionBox in a context, it is compared to other toolkits to examine its interaction features in comparison to other tools. The comparisons show that JunctionBox provides a richer set of interaction options than the other tools. By providing a rich set of interactions, JunctionBox opens the door to greater creativity in designing multi-touch musical interfaces.

JunctionBox is also explored via practice-based research. During my research, I have created and performed with a variety of interfaces that I built with JunctionBox. These interfaces range from live performance interfaces to controls for an interactive installation. The variety of interfaces shows the flexibility inherent in the design of JunctionBox. In addition, these interfaces serve to show the creative interface possibilities that JunctionBox affords.

Finally, research into the design and implementation of JunctionBox led to the de-

velopment of a series of design principles that can be applied to toolkits that aspire to balance features and creative freedom. The design principles are variations on tolerance. Tolerance for allowing developers to use their own creativity in designing and building musical interfaces.

Acknowledgements

First I want to thank my supervisors Sheelagh Carpendale and David Eagle. Their feedback and support was invaluable. I also want to give a special thanks my friend and collaborator Adam Tindale for supporting my research in so many ways. Finally, I want to thank the members of the University of Calgary Interactions Lab who were my friends and supporters throughout my research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *I wanted to give the musician a great deal of power and generality in making the musical sounds, but at the same time I wanted as simple a program as possible; I wanted the complexity of the program to vary with the complexity of the musician's desires. If the musician wanted to do something simple, he or she shouldn't have to do very much in order to achieve it. If the musician wanted something very elaborate there was the option of working harder to do the elaborate thing. The only answer I could see was not to make the instruments myself – not to impose my taste and ideas about instruments on the musicians – but rather to make a set of fairly universal building blocks and give the musician both the task and the freedom to put these together into his or her instruments.*
>
> –**Max Mathews** [97]

Music-making is one of humanity's most ubiquitous and creative activities and throughout human history, people have used a variety of tools for creating music. Each new tool, from the earliest bone flutes to electric guitars, represents a step in the development of music technology. Humanity's latest tool, the computer, is a relatively recent addition in the development of music technology. As with earlier tools, the computer has been used, from its inception, to make music. In the history of using computers for music, the way that people interact with computers has changed over time as new forms of human input are introduced. Multi-touch input is a recent and nearly ubiquitous way for humans to interact with computers of different kinds including phones, tablets, monitors,

and tables. Networking is another ubiquitous aspect of computers that allows for interactions between computers. With so many multi-touch, networked computers around, it is natural to harness their capabilities for music-making. A thorough investigation of multi-touch and networking in a musical context can unlock their combined musical potential.

In the quote that opens this chapter, Max Mathews was referring to his MUSIC III software created in 1960. MUSIC III is significant because it introduced the unit generators model in which basic signal processing units in digital audio like sine waves, envelopes, and filters that are very simple on their own but can create very complex sounds when used in combination. The unit generator model proved to be very successful and its usefulness continues to the present day in the form of countless software projects that continue to use the model. Using the success of the unit generator model as a starting point, my research draws from the foundational notion of *universal building blocks* and applies it to multi-touch interactions in order to enable a wide range of creative possibilities for building musical interfaces. The research question that I address in this thesis is how to apply the *universal building blocks* model to multi-touch interactions and networking in the form of a software toolkit.

This chapter begins with some brief background on interacting with computers to make music (1.1). This background sets up the motivation behind my research (1.2) that is narrowed by a description of the scope of the research (1.3). This is followed by the central question of my research (1.4) setting the stage for a description of the challenges I faced during my research (1.5). This leads to a description of the methodology I used to engage in the research (1.6) followed by a list of the contributions from this research (1.7). This chapter ends with an overview of the structure of this thesis (1.8).

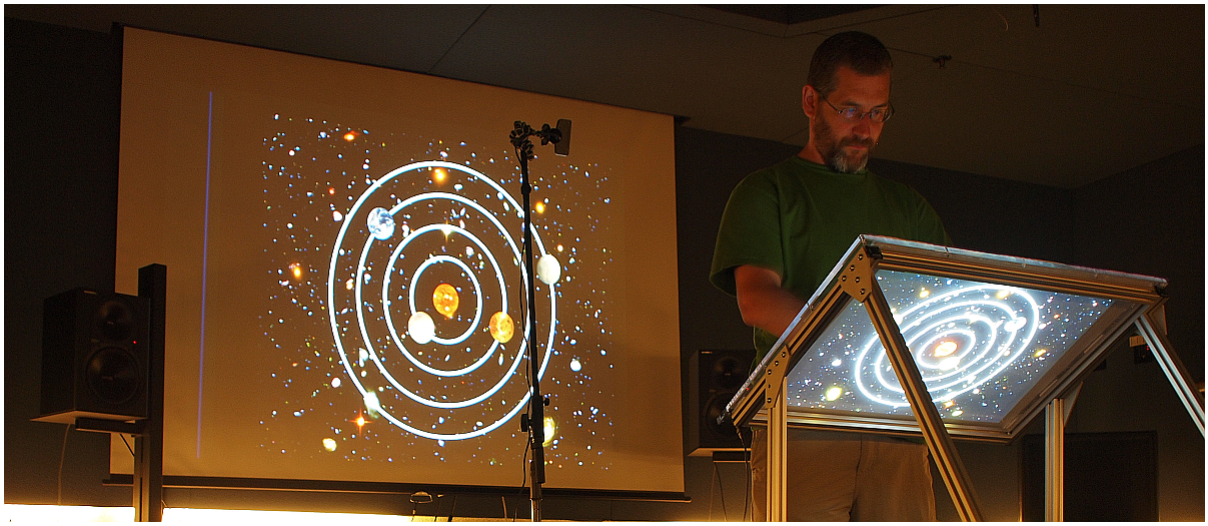


Figure 1.1: The author of this thesis playing a multi-touch, networked musical instrument of his own design.

1.1 Interactive Computer Music

As a form of human-computer interaction, multi-touch is not only for music but is a generalized kind of interaction that can be used in a variety of contexts on a variety of computing systems. Throughout its history, computer music has taken new ways to interact with computers, from keyboards and mice to multi-touch, and used them in a musical context. The history of computer music began with interactions on generalized computing systems. When Max Mathews created the first audio synthesis language, MUSIC, in 1957, he did it on an IBM 704 computer that was not designed and built for creating music [97]. Figure 1.2 shows an IBM 704 in action [19]. MUSIC set a precedent for a thread of computer music research that took general-purpose computers like the IBM 704 (or current multi-touch devices) and used them for making music.

In order to interact with the 704 and similar early computing systems, composers and musicians would program their pieces using punch cards. This process was not real-time in that a composer could not get an immediate result. Instead the entire piece could only be heard once all of it was programmed and played back. This lack

of real-time feedback meant that these early computers were not performable as live musical instruments. Increased computing power has led to the creation of programming environments for music [72, 89, 117] that have brought computer music from the days of punch cards to the possibility of real-time interactions. The ability to handle real-time interactions has allowed computer-based music systems to go beyond compositions to something performable. Multi-touch devices appeared after the development of real-time computer music and so they are well positioned to become a part of performable musical systems.

Figure 1.2: The IBM 704 computer: not built specifically for making music.

Another development in computing, aside from real-time interaction, is the ability for computers to interact with each other via networking. Just as computers were becoming

more widely available, the ability to connect them with wires (and eventually wireless) became easier. Groups like the League of Automatic Music Composers (that later became the Hub) [11] began, in the late 1970s, to use small relatively inexpensive computers networked together to create music as a group. With the possibilities afforded by networking, the notion of an instrument expanded to be more like a musical system in which individual computers and musicians could be considered parts of a single instrument.

1.2 Motivation

When a new human-computer interaction, like multi-touch, is developed or when an interaction begins to become ubiquitous, it is natural that the new technology will be used in a musical context. From phones to tablets to larger devices like tables and walls, various kinds of multi-touch-enabled hardware are readily available as potential interfaces for controlling music. The process of taking multi-touch interactions and using them for various kinds of musical control is known in computer music as *mapping*. The following quote from Hunt et al [49] explains the importance of mapping in electronic instrument design (in contrast with traditional acoustic instruments):

> *The interface is usually a completely separate piece of equipment from the sound source. This means that the relationship between them has to be defined. The art of connecting these two, traditionally inseparable, components of a real-time musical system (an art known as* ***mapping****) is not trivial.*

In the preceding quote, the authors state that the interface is "usually a separate piece of equipment". However, regardless of whether the human interface is a separate piece of equipment, it is the case with computer-based instruments that they cannot operate without some sort of mapping. Computers are calculating and logic devices that can turn human (analog) input into numbers. They can then output numbers that are converted

to analog signals destined for speakers. When computers are involved, mapping is a requirement, not something that is "usually necessary". This being the case, it is vitally important to think about the necessity and the flexibility inherent in mapping when designing toolkits for building multi-touch instruments.

By supporting computer networking, a toolkit can offer an even greater degree of flexibility. Networks connect computers together, allowing musical systems to go beyond a single computer to two or even more computers. Multiple computers can be connected as separate musical instruments or they can be connected to form a single musical instrument. With networking, computers can be connected in local networks (generally in the same room) or they can be connected over the internet. Regardless of whether instruments span a local network or the internet, musicians should be able to interact with computers connected over the network just as easily as they can with computers in front of them.

Multi-touch hardware requires software in one of two broad categories: 1) applications for some specific purpose and 2) libraries (or toolkits) for building applications. Applications generally make it easy to accomplish their specified task but, at the same time, their specificity limits their flexibility. On the other hand, software libraries require more effort, via programming, but they have tremendous flexibility. My research is designing and building software toolkits because I believe that programming, more than any application, can leverage the flexibility of software to allow for creative freedom while, at the same time, providing important functionality. The motivation for my research is to enable creative freedom by building a toolkit that combines multi-touch and networked music while at the same time allowing musicians to build the interface that suits their own creative whims.

1.3 Scope and Audience

My research is at the intersection of three areas of computer music research: 1) multi-touch music interfaces, 2) networked music, and 3) software toolkits. Figure 1.3 shows the intersection of the three areas in graphical form.

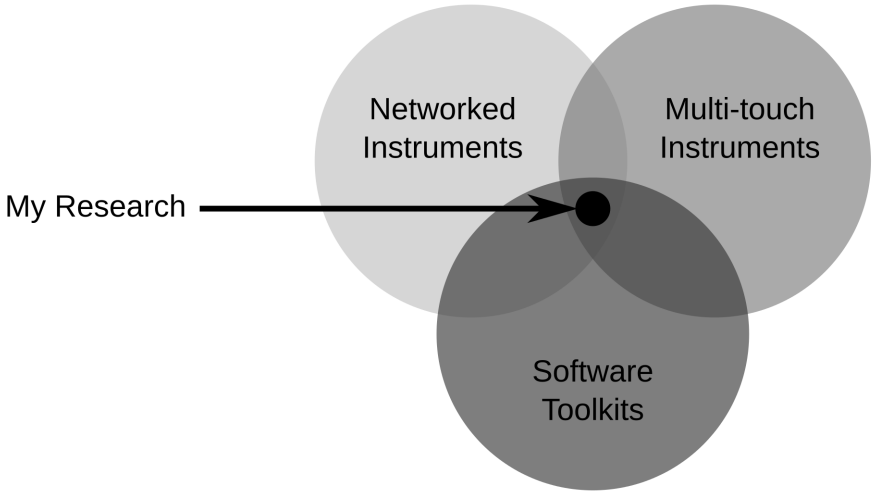


Figure 1.3: The scope of the research presented in this thesis.

The scope establishes the audience for my research: computer music researchers and instrument builders who want to program their own multi-touch musical interfaces. This audience inspires my research because I believe that putting programming tools into the hands of creative people will enable them to build the interfaces that they imagine rather than imagining what they can do with existing interfaces.

1.4 Research Question

To enable builders to program the interfaces that they imagine, I based my research on the foundation of Max Mathews' model of *universal building blocks* for programming audio. Just as Mathews intended his universal building blocks to enable freedom for builders to create audio programs from the simple to the complex, my goal is to enable

creativity and freedom for building multi-touch interfaces. This goal leads to the central question for my research:

> **Can Max Mathews' *universal building block* model for programming audio be applied to programming multi-touch, networked interactions?**

1.5 Challenges

To answer the research question of applying the *universal building block* model to multi-touch interactions, I needed to address a number of challenges. During the course of answering this research question, I addressed the following challenges:

1. **Find the universal building blocks inherent in multi-touch input and create a unit interaction model**

   Multi-touch devices afford a variety of interaction possibilities. The challenge for my research with these devices was to identify the most basic multi-touch interactions and to use these to create a *unit interaction* model that represents a set of universal building blocks that can be used in any combination to create a range of musical interfaces from the simple to the complex. Since my research is investigating multi-touch in a musical context, each of the interaction building blocks needs to be easily mappable to musical control. This means that the unit interactions are not just basic multi-touch interactions but are also a set of mappable unit interactions.

2. **Show that the unit interaction model has been successfully applied by building a toolkit that reifies the model**

   The unit interaction model can be reified by building a software toolkit based on the model, allowing the success of the model to be evaluated. In order to evaluate

the success of the unit interaction model, I take two approaches.

(a) If the interactions in the unit interaction model have been successfully identified as unit (fundamental) interactions, it should be possible to compare my toolkit to existing mapping tools to determine whether my toolkit affords more mappable interactions.

(b) A software toolkit can show its value by being useful in real situations like building interfaces and using them in performance situations. If the unit interaction model has been successfully applied, the toolkit based on the model should be usable for actual performances.

3. **Distill the research into a set of design principles**

   The process of applying the unit interaction model in the form of a software toolkit should itself yield some lessons for how to apply the notion of universal building blocks to multi-touch mapping toolkits.

1.6 Methodology

The methodology employed in this thesis is a form of practice-based research inspired by Sullivan [106] and his call for emphasizing the possible in the act of creating:

> *If an agreed goal of research is the creation of new knowledge, then it should be agreed that this can be achieved by following different, yet complementary pathways. What is common is the attention given to systematic and rigorous inquiry, yet in a way that emphasizes what is possible, for to create and critique is a research act that is very well suited to arts practitioners, be they artists, teachers or students.*

Proceeding with Sullivan as an inspiration, I applied his methodology in a way that suits my research interests and musical practice. The methodology has five components: study, building, analysis, performance and principles. There is a clear pathway which starts at study, moves to building, branches into analysis and performance, and ends with principles. Analysis and performance represent Sullivan's complementary pathways as shown in Figure 1.4. Both analysis and performance lead to principles that represent what Sullivan calls "the creation of new knowledge".

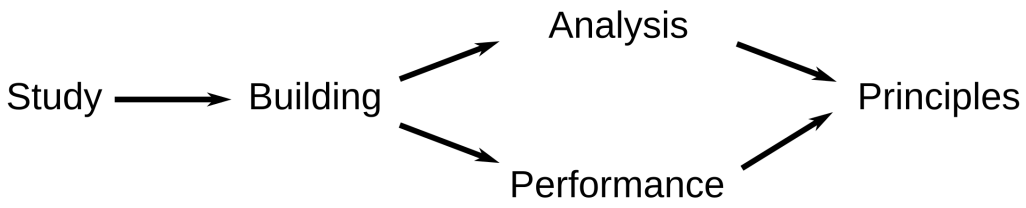


Figure 1.4: The five-component methodology used for my research.

1. **Study**

   To understand the state of multi-touch, networked musical instruments and software toolkits, I studied the literature to find relevant related work. By studying related multi-touch instruments, I was able to understand the kinds of interactions that they supported. Networked instruments showed the importance of allowing for a variety of networking configurations. Finally, in looking at related software toolkits, I was able to determine what interaction features they did and did not have. In particular, I selected three closely related tools as a basis for determining the feature requirements for my research.

2. **Building**

   Having understood the related work, I designed and programed a software toolkit for creating multi-touch, networked musical interfaces. The development process

for the software involved programming and testing features to ensure that they met the feature requirements as determined in the Study component.

3. **Analysis**

   To determine whether the toolkit offered more interactions than closely related tools selected in the Study component, I employed a comparative analysis to demonstrate that the toolkit offers a greater number of both low-level interactions and combinations of interactions, showing that the toolkit is a superset of the related tools.

4. **Performance**

   To show that the software is usable for real musical performances, I designed and built a series of musical interfaces and used those interfaces in my own performances. For each interface, I started with concepts for both the graphical part of the interface and the interaction part. Once I had established my concept, I programmed the interfaces with the software. The performance scenarios for these interfaces varied from demos to live performances to installations.

5. **Principles**

   Based on the comparative analysis and musical performances, I derived a set of design principles for building multi-touch, networked software toolkits that have a high degree of creative flexibility. The process of deriving the design principles distilled the implementation details down into usable guidelines for building multi-touch, networked creative coding toolkits. Design principles are important because, once distilled from implementation and use, they can outlive changes in technology in their usefulness as a contribution to knowledge.

1.7 Contributions

By applying the methodology described in Section 1.6, I met the challenges in Section 1.5, leading to a number of research contributions.

1. **A unit interaction model for multi-touch**

   By examining the literature on multi-touch instruments past and present and by using various multi-touch mapping tools, I developed a model for unit interactions that can be used in a variety of combinations. This model represents a set of universal building blocks for multi-touch interactions. Each of these interactions is in itself basic but they can be combined to create many kinds of interface controls, enabling a range of interface styles from the simple to the complex.

2. **A toolkit that reifies the unit interaction model**

   The unit interaction model was reified by building the JunctionBox toolkit, described in Chapter 4. All of the multi-touch unit interactions implemented in JunctionBox are mappable to musical control and fully networked. The process of developing mappable interactions in JunctionBox represents the application of the model to something that is both comparable to other tools and usable in musical performances. The full set of mappable interactions in the toolkit is described in detail in Chapter 4, Section 4.3. To show that my unit interaction model was successfully applied, I used the following evaluation approaches to determine whether I met my goals of identifying unit interactions and then making them usable for musical performances.

   (a) In order to evaluate the success of JunctionBox as a set of unit interactions, I compared it to similar mapping tools. The success of the unit interaction

model was measured by comparing the absolute number of mappable interactions offered by JunctionBox with the number offered by the tools under comparison. The comparisons were based strictly on the number of mappable interactions and not on the full feature set in JunctionBox or the other mapping tools since their non-interactions features do not overlap. The analysis showed that my unit interaction model was able to identify more mappable interactions than the other tools. Details of the analysis are in Chapter 5.

(b) To show the viability of my model, I built a series of interfaces with JunctionBox and used them in musical performances. These interfaces show my interest in using the toolkit for my own but they also serve as examples of what the toolkit can accomplish. At the same time, my work creating interfaces with the toolkit shows its ability to perform in real musical situations. My interfaces and performances are described in Chapter 6.

Beyond interaction mapping, the toolkit has a number of what I consider special features include saving the state of interface controls after interactions have changed them, having inheritable interactions between different interface controls, recording interactions for later playback, and managing connections and messages among networked computers. The special features strengthen the overall contribution offered by the toolkit since they are not offered by similar tools. These special features are detailed in Chapter 4, Section 4.4.

3. **A set of design principles**

The final contribution is a set of design principles that I derived from working on and with the toolkit. The design principles represent the lessons that I learned in building a toolkit that gave me the freedom to design the interfaces according to my own creative whims. Their significance lies in there use as guidelines for building

creative coding toolkits that are independent of implementation. This contribution is described in Chapter 7.

1.8 Overview

The following are chapter descriptions for the rest of this thesis:

Chapter 2: Musical Context

Chapter 2 provides musical context for the work presented in this thesis. First, computers as instruments are placed in the context of advancing music technology including some background on why computers are significant as music technology and some history of relevant instruments and their designers (Section 2.1). This is followed by a discussion of experimental music performance with descriptions of the work of music groups that embrace both technological and musical experimentation. (Section 2.2).

Chapter 3: Related Work

Chapter 3 includes work related to my research and is broken into three sections according to the scope of the research. See Section 1.3 in this chapter for details about the scope. The first section describes related work in multi-touch instruments (Section 3.1). The second section describes related networked instruments (Section 3.2). The third section describes related software toolkits (Section 3.3).

Chapter 4: JunctionBox

Chapter 4 describes the JunctionBox toolkit in detail and is broken into three sections. The first section presents the fundamentals of how JunctionBox enables multi-touch interaction mapping and networking (Section 4.2). The second section presents a full list of mappable multi-touch interactions (Section 4.3). The third

and last section describes the special features built into JunctionBox that go beyond multi-touch interactions (Section 4.4).

Chapter 5: Comparative Analysis

Chapter 5 is a comparative analysis of the mappable multi-touch interaction options offered by Junction relative to similar mapping tools. The chapter is broken into three sections, one for each similar tool. The fist section compares Control (Section 5.1). The second section compares TouchOSC (Section 5.2). The third section compares Lemur (Section 5.3).

Chapter 6: Interfaces and Performances

Chapter 6 describes a series of music performance interfaces that I built using the JunctionBox toolkit. Each section features a different interface that was used in one or more performance situations. The performance situations are described along with the rationale for the visual design and the mappable multi-touch interactions afforded by each interface.

Chapter 7: Design Principles

Chapter 7 examines the design principles that emerged during my research. The principles are described in detail along with explanations for their derivation and their application to other toolkits.

Chapter 8: Conclusions

Finally, Chapter 8 summarizes the research contributions (Section 8.1) and describes future work in extending the research presented in this thesis (Section 8.2). The thesis ends with some closing remarks (Section 8.3) and a short coda (Section 8.4).

Chapter 2

Musical Context

Composers are now able, as never before, to satisfy the dictates of that inner ear of the imagination. They are also lucky so far in not being hampered by aesthetic codification–at least not yet! But I am afraid it will not be long before some musical mortician begins embalming electronic music in rules.

–**Edgard Varèse** [114]

This chapter is included to set my research in musical context. In this chapter, I provide a musical context for my research in two ways: 1) by placing computers as instruments into a larger context of advancing music technology and 2) with an overview of ensembles that experiment with both music and technology. These two contexts represent the two threads of my research as a developer of new musical systems and as a performer and improviser. The first section (2.1) provides relevant context on advancing music technology and the second section (2.2) discusses the development of experimental performance practices that combine musical improvisation with experiments in music technology. The last section (2.3) provides a short summary.

Max Mathews wrote the first digital computer music program in 1957. As computer music was getting started with Max's program, ideas about how music can be composed and performed were changing, providing an evolving artistic context for the development of computer music. Throughout the 1950s, composers like John Cage, David Tudor, Morton Feldman, Earle Brown, and Christian Wolff, among others, were challenging both the compositional process and performance practice in what Michael Nyman calls "experimental music" [79]. At the same time that experimental music was developing,

the notion of what could be considered a musical sound was changing, especially with the development of acousmatic music [102] in which any sound could be used to create a musical work. In 1948, Pierre Schaeffer created his famous acousmatic piece, *Étude aux chemins de fer*, using nothing but phonograph recordings of trains as his sound materials. As Edgard Varèse predicted in as far back as 1936 [114], the very definition of what can be considered music had been liberated from earlier, more restrictive definitions. This liberation provides a context in which the very definition of music is not entirely clear.

The history of computer music has been a parallel processes of experimental development. Instead of trying to redefine the musical process with traditional instruments, computer music has taken new developments in computing technology and used them for musical purposes. Each new computer technology, from networking to various forms of human input (keyboard, mouse, multi-touch to name a few) has redefined the context in which computer music is made. My research exists in this experimental musical context: one in which everything from the processes of composition to the use of sound materials to the nature of performance itself are not so clearly defined. At the same time, my research exists in the context of every-changing ways for people to interact with computers.

2.1 Advancing Music Technology

Technology has always played an important role in making music. In this age of ubiquitous computers, discourse about technology tends to focus entirely on computers. However, aside from the human voice, all musical instruments from drums, to flutes, to violins are all examples of music technology. Any instrument, whether a computer is involved or not, is just another form of music technology. Figure 2.1 shows one of the earliest known pieces of music technology: a set of bone flutes found in China. It is not difficult

to imagine that these bone flutes were not the first created, but that they are a result of research and experimentation. The same can be said of any subsequent instrument. A full history of the development of music technology is beyond the scope of this thesis. However, it is important to point out that music technology has a long history and that all musical instruments, including computers, are part of that history.

Figure 2.1: Examples of music technology. These playable bone flutes are between 7000 and 9000 years old. [9]

2.1.1 Computer(s) as Instrument(s)

By thinking about all musical instruments as a form of music technology, it is easy to see that computers are just the latest iteration in the long human tradition of using technology of any kind to make music. However, there is an important difference between computers as instruments compared to older acoustic instruments. The fact that acoustic instruments have been used by humans for making music for far longer than computers means that there are whole sets of traditions and a considerable amount of history that informs how acoustic instruments are used. Compared to the thousands of years that have gone into the development of acoustic instruments, computer-based instruments are still relatively new. Coupled with a rapid evolution in how humans interact with

computers, the computer as instrument is still in a relatively experimental stage. In essence, the potential for computers as instruments is still being explored.

Max Mathews understood the potential for computers as musical instruments when he made the following assertion in 1963:

> There are no theoretical limitations to the performance of the computer as a source of musical sounds, in contrast to the performance of ordinary instruments. [71]

This lack of limitations means that a computer can generate an almost unlimited number of sounds. This also means that a computer can also act as multiple instruments at the same time. With advances in computer networking, multiple computers can be combined into a single instrument. This range of possibilities, from the ability to make any sound to the ability to combine them in many ways, makes computers unique in the history of music technology.

Beyond sounds and networking, computers have other unique properties. Computer-based instruments have, by their nature, an inherent energy as long as they have a power source. In contrast, acoustic instruments are human powered in that they only generate sound when the performer is putting energy into them. In addition to having a performative energy, computers are also programmable, giving them the capability to be autonomous in terms of both energy and decision making. That is:

$$\textbf{energy} + \textbf{programmability} = \textbf{autonomy}$$

The potential for autonomy gives computers a range of performative possibilities. At one extreme, a computer can completely take over the role of performer. At another, a computer can be programmed to only generate sound when a human performer initiates a musical gesture. In between these extremes lies interesting opportunities to have a balance between the human as performer and the computer as performer.

The notion of autonomy has given rise to a kind of music that is unique to computers. Algorithmic music is a kind of music in which the computer makes decisions about musical output in an autonomous or semi-autonomous fashion. In fact, there is a spectrum of algorithmic music from entirely autonomous to what the composer Curtis Roads calls "automation with interaction" [96]. In particular, Roads defines a specific kind of interaction with algorithms in live performances as:

> ...intense real-time interaction experienced in working with a performance system onstage, where the emphasis is on controlling an ongoing musical process and there is no time for editing.

In this scenario, the person and the computer work together to produce the musical output in real-time as opposed to developing music in a studio with editing. When an equal balance is struck between human and computer, the human performer becomes more like a conductor, giving instructions to the computer that change the sound during performance.

On the subject of human control versus computer autonomy, Nilsson [76] makes a distinction between *playing* instruments and *controlling* them. Nilsson's distinction can be understood as the difference between acoustic instruments which have a strong action-sound link and electronic and computer instruments that afford different linkages between action and sound. Acoustic instruments have a strong action-sound link because the gestures used by a performer are in direct proportion to the sound result of the gesture. With electronic or computer-based instruments, this linkage between gesture and sound is possible but not strictly necessary. For situations in which the action-sound linkage is weak, Nilsson uses the term *control* to refer to this kind of interaction between human performer and computer. My own performance practice involves me *controlling* semi-autonomous musical processes. The interfaces that I discuss in Chapter 6 are controllers

that allow me to conduct a musical performer. In my case, the performer is a computer.

Figure 2.2: Léon Theremin playing his eponymously-named instrument.

2.1.2 Instrument Designers

With so many changes in technology, particularly with the rapid change in computing interfaces, it is the role of the instrument designer to take new technologies and to find ways to design them to make music. The increasingly widespread availability of electricity in the early 20th century gave rise to a new class of instruments with new kinds of sounds. Léon Theremin patented his eponymously-named theremin instrument in 1928. The theremin was one of the first purely electronic instruments and it created a unique sound that no acoustic instrument could make. Another electronic instrument, the ondes Martenot, invented by Maurice Martenot in 1928, featured similar electronic sounds with a piano-like keyboard. Hugh le Caine's electronic sackbut, from 1948, was another keyboard-based electronic instrument with a unique sound. These instrument showed the potential for using electricity as a medium for musical expression.

The potential for musical expression with electronic instruments took a great leap with the appearance of modular synthesizers. In the 1960s, Robert Moog developed a series of modular synthesizers in which the electronics that generated the sounds could be configured by the performer. Unlike, the earliest electronic instruments, the modular synthesizer allowed for experimentation with sounds. During this time, Don Buchla worked on similar modular synthesizers such as the one shown in Figure 2.3.

Figure 2.3: A Buchla analog synthesizer with knobs, buttons, and cables for changing the basic sound of the synthesizer.

Moog's modular synthesizers, with their tremendous control over sounds, found their way into a variety of recordings from the Beatles to Wendy Carlos' *Switched on Bach* album from 1968. On that album, Carlos plays traditional Bach compositions on the Moog synthesizer. Musicians like Keith Emerson of Emerson, Lake, and Palmer and Rick Wakeman of Yes used a portable version of Moog's synthesizer, the Minimoog, throughout the 1970s. By allowing performers to have fine-grained control over their instruments, the synthesizers of Moog and Buchla opened up new ways to experiment with sounds at the same time that the instruments designers experimented with the synthesizers themselves.

Figure 2.4: The Yamaha DX-7 synthesizer enabled experimentation with digital sounds.

The notion of control over sounds has moved from the pure electronics of the early synthesizers into the digital sound realm of the computer. The Yamaha DX-7, based on FM synthesis research by John Chowning [15] in the late 1960s and early 1970s, brought new kinds of sounds and new controls in the form of a digital keyboard in the 1980s. As a digital synthesizer, the DX-7 gave musicians new ways to experiment by selecting the fundamental algorithms that generated the sounds.

Figure 2.5: The reacTable modular synthesizer is controlled with blocks and multi-touch.

The reacTable instrument, appearing in 2005, (see Figure 2.5), borrows many of the same concepts from the early modular synthesizers and puts them into a new kind of package with new technologies. Instead of keyboard-like controls, the reacTable is a combination of human input, via the placement of blocks or multi-touch, and virtual controls that are projected onto the surface of the instrument. Like earlier modular synthesizers, the reacTable has the ability to fundamentally control sounds. The reacTable modular

synthesizer was played by Damian Taylor on tour with Icelandic singer Björk in 2007 [98]. The use of the reacTable in Björk's band shows that interest in having fundamental control over instruments is still strong. With their ability to produce any kind of sound coupled with control over any sound parameters, computers provide instrument designers with a virtually infinite canvas with which to create new kinds of instruments.

My own work on advancing the use of multi-touch for musical control is a continuation of the lineage of taking new kinds of computer input and using them for music. As an instrument designer, I am interested in both making multi-touch as useful as possible in a musical context and in keeping the spirit of experimentation from earlier modular instruments alive. By giving musicians control over their multi-touch interactions, I am bringing that experimental spirit to the control of human input.

2.2 Experimental Music Performance

In order to show an evolution of influences that are relevant to my research, I have selected a series of musical ensembles that form a thread that provides a context for my own work, both technically and musically. All of these ensembles have two things in common with my own musical practice: 1) all of the ensembles make improvisation an important part of their music and 2) each of the ensembles takes an experimental approach to the technology that they use to create their music. A constant theme in experimental music is the adaptation to new technologies. Musicians do not perform outside of a musical or a technological context. As musical styles change, musicians adapt and respond to those changes and the same is true of technology. Musical styles can remain constant across new technologies. However, new technologies fundamentally change the ways that music is performed.

2.2.1 Musica Elettronica Viva (1966–Present)

Musica Elettronica Viva (MEV) was founded by American composers Allan Bryant, Alvin Curran, Jon Phetteplace, and Frederic Rzewski in Rome in 1966. The group was initially formed to showcase the compositions of each of its members. However, early in its existence, MEV moved away from fixed composition and into improvised performances. According to Rzweski:

> We began by performing compositions by ourselves and others which involved the use of electronic sound produced in real time, or "live" electronic music. In the summer of 1967 we began to work more with improvisation and less with determinate structures. [101]

By improvisation, Rzewski is not referring to the "chance" music from composers like John Cage in which dice, the I Ching, or other random process is used to compose music. Rather, he is referring to music that is created by musicians collectively in the course of performance. That is, musicians compose their music during performance in relation to their own previous sounds and the sounds made by the other performers. Improvisation is essentially experimental music since the end result is not known in advance.

MEV is an experimental music group in another sense as well: they experimented with the use of electronics for music. For performances of their collective improvisation piece, *Spacecraft* in 1967, the members of MEV used an amplified glass plate, an amplified tin can, amplified voice, and a Moog synthesizer controlled by brain waves in addition to acoustic instruments like saxophone, trumpet, and thumb piano [1]. Any performance by the group would feature both musical improvisation as well as experimentation with instruments old and new.

Figure 2.6: Members of Musica Elettronica Viva, from left, Frederic Rzewski, Richard Teitelbaum, and Alvin Curran [21]. Note the trumpet in the foreground and the Moog synthesizer in the background.

2.2.2 The League of Automatic Music Composers (1978–1983)

The League of Automatic Music Composers was formed in 1978 by David Behrman, Rich Gold, John Bischoff, and Jim Horton. The four members were involved with the Center for Contemporary Music (CCM) at Mills College in Oakland, California. At the time of the formation of the League, the Bay Area of California had a fertile experimental music scene [11]. At the same time, small personal computers were beginning to become available in what later become the Silicon Valley. The League were early adopters of small personal computers and the members had a keen interest in using these in an experimental musical context. The name, the League of Automatic Music Composers, was chosen by the group to reflect their experimental inclinations:

> It also sought to convey the artificial intelligence aspect of the League's activities as we began to view half the band as "human" (the composers) and half "artificial" (the computers). As stated in our concert program, "the League is an organization that seeks to invent new members by means of its projects". [10]

Members of the League were inspired by the chance music of John Cage and others and they sought to use computers to bring chance operations to their performances. Unlike Cage, the League also explored improvisation in their performances. In their performance practice, the League layered experimental music practice on top of technological experimentation.

Figure 2.7: The League of Automatic Music Composers in 1981: Tim Perkis, John Bischoff, Don Day, and Jim Horton.

2.2.3 The Hub (1985–Present)

The Hub grew out of the League of Automatic Music Composers with some of the same members including John Bischoff, Tim Perkis, Chris Brown, Scot Gresham-Lancaster, Mark Trayle, and Phil Stone. The approach to music-making was carried over from the League with an emphasis on exploring new technologies. The name Hub refers to both the group and to their technical setup in which they used a central hub of musical data that is shared by the six members. Advances in computer networking allowed the Hub to connect their individual computers to this central hub, providing a new way for the members to share their improvised musical data. One of the founders, Scot Gresham-Lancaster describes the ethos behind the Hub [41]:

> Interactive electronic music constitutes a continuing story of the ingenious use of technologies in unique and unconventional ways.

The Hub adapted to new technologies continually, trying varying combinations of computing and networking, including an early attempt to use the internet for a musical performance. With each new technology, the Hub continued to experiment both musically and technologically.

Figure 2.8: The Hub: Chris Brown, Scot Gresham-Lancaster, Mark Trayle, Tim Perkis, Phil Stone, and John Bischoff.

2.2.4 Sensorband (1993–2003)

Sensorband was formed in 1993 by Edwin van der Heide, Zbigniew Karkowski, and Atau Tanaka. The group took a very different approach to human input than either the League or the Hub. Instead of keyboards and mice, Sensorband used ultrasound, infrared, and bioelectric sensors for human input. Sensorband performances had a strong theatrical element since as Atau Tanaka states [109]:

> Sensorband's projects center around the theme of physicality and human control/discontrol in relation with technology.

Each member had his own specially-designed sensor as part of his instrument. Edwin van der Heide used a MIDIconductor that used ultrasound signals to determine the position of his hands as well as the distance between them. Zbigniew Karkowski used

infrared sensors to detect the position of his arms in the space around him. Atau Tanaka played his BioMuse system [108] that used an electromyogram (EMG) sensor to detect his muscle tension. In each case, the sensor input was digitized, enabling the performers to translate their physical gestures into musical gestures.

Figure 2.9: Sensorband in performance with, from left, the MIDIconductor, infrared sensing in space, and the BioMuse. [107]

Sensorband performances had a strong improvisational component as the group experimented musically and technologically with their use of sensors. By experimenting with sensors, Sensorband showed that computers can be controlled with virtually any kind of human input.

2.2.5 The Princeton Laptop Orchestra (2005–Present)

Laptop orchestras are ensembles made up of several performers and their laptops. Generally, each performer has their own speaker or speakers for their laptop sounds. Laptop orchestras are generally open about human input to the laptop itself including everything from the laptop keyboard and mouse to attached devices like joysticks and drawing tablets. The first laptop orchestra was the Princeton Laptop Orchestra or PLOrk [113]. Figure 2.10 shows a PLOrk performance. Note the laptops with individual speakers and the performers sitting on pillows.

In his paper, *Why a laptop orchestra?* [112], Dan Trueman, one of the founders of PLOrk, makes the following point about the nature of the instruments they used:

> ...it is important to point out that the goal isn't necessarily to create finished instruments that remain with us for generations, but rather to develop a

Figure 2.10: The Princeton Laptop Orchestra (PLOrk) performing.

> performance practice where instrument building itself plays a central role. This is one of the great enticements of building digital instruments; we don't have to spend a year carving up several pieces of wood to explore a new acoustical design.

For performances, PLOrk would feature laptop performers, optional soloists, and a conductor. The performances were often a structured improvisation in which the performers have some freedom to change their sound but are still expected to follow the overall structure of the piece as conveyed by the conductor via hand signals.

2.2.6 The Stanford Laptop Orchestra (2008–Present)

The Stanford Laptop Orchestra or SLOrk [116] was initiated by Ge Wang when he moved from Princeton, where he co-founded PLOrk, to Stanford University. In early 2008, he created a "pre-laptop orchestra" course in which the technology used by SLOrk was to be built. Along with about 30 other people, I was part of this course. The main objective and the most work was put into creating custom speakers that would be used by each member of the orchestra since, as with PLOrk, the idea behind SLOrk was that each member would have their own instrument consisting of a laptop, some kind of human

input device, and a speaker.

The speakers needed to be mobile so that they could be transported to performances as needed. In addition, the speakers needed to have a non-directional sound. Speakers normally output their sound from the front but, for SLOrk, the speakers needed to output sound in multiple directions since the intention was for each member to have their own sound field that radiates from their instrument. While there are commercial speakers like this available, we decided to build out own speakers. To build them, we took inexpensive, lightweight wooden salad bowls and drilled holes in them for holding small car speaker drivers (the part that actually generates the sound), six in total. Figure 2.11 shows the bowls before the drivers were attached.

Figure 2.11: Making speakers from salad bowls. The holes are for mounting car speaker drivers. [115]

Once the drivers were attached, we attached a plug strip for audio cables to allow the signal to get into the speakers along with power connectors for amplifiers that were stored inside of the speakers. Once everything was set inside the speaker, we put a custom-fitted wooden plug on the bottom of each speaker. In addition, we attached small handles to make it easier to carry the speakers. Figure 2.12 shows a finished speaker with salad bowl, drivers, audio cable connections, power connection, and handle.

Figure 2.12: A finished SLOrk speaker with six car speaker drivers, audio cable connections, power connection, and carrying handle. [115]

The do-it-yourself (DIY) nature of the speakers is part of the overall experimental nature of SLOrk. Beyond the creation of new music technologies like the speakers, SLOrk was a vehicle for determining what laptops can do in the context of a 20-person orchestra. When composing for laptop orchestra, a composer needs to thing about many possibilities including the sounds to use (remembering Max Mathews observation about computers producing any kind of sound), how the sound(s) will be controlled, and whether each member of the orchestra will have the same sounds or whether they will have different sounds [104]. Figure 2.13 shows a SLOrk performance from April 29, 2008 at Stanford University in which we controlled sounds by tilting out laptops.

2.2.7 The Aspect Ensemble (2012–Present)

Aspect is a music ensemble featuring myself, Simon Fay, and Aura Pon. I initiated the formation of this ensemble so that it would serve as a musical vehicle for the research of its members. I chose the name Aspect because it was my intention to make visuals an important part of our performances. Instrumentation varies, but Aspect always features digital instruments of some kind. Generally, I played with a multi-touch tablet, Simon

Figure 2.13: The Stanford Laptop Orchestra (SLOrk) performing, including the author of this thesis, second from the right.

played electric guitar, and Aura played oboe. In other cases, as with Aura's piece, *Concordia Discors*, each of us played an iPhone while getting instructions for the piece from an animated digital score. For another performance, we improvised a score for the 1903 silent file, *Alice in Wonderland* with myself on tablet, Simon with a laptop instead of guitar, and Aura on oboe.

Figure 2.14: Aspect rehearsing with a visual score. We see the same score on a laptop in front of us.

In forming Aspect, I saw the potential of the multi-touch tablet computer in a performance context and decided to use tablets to control my part of Aspect's music. Tablets had shown their usefulness for making music before I started Aspect. In 2010, Apple Inc.

introduced the iPad tablet, a device that offered direct touch interactions with graphics that was similar to the iPhone. However, the iPad offered a larger screen (and touch surface) and more power. Immediately after its release, musicians were using it to make music. The band Gorillaz, while on tour in 2010, recorded and released an album, The Fall [39], created entirely with an iPad.

My initial reasons for choosing a tablet as my interface were portability and the ability to put my software, JunctionBox, onto tablets. Via my performances with Aspect, I learned that tablets also suit my own performance style. In contrast to the obvious theatricality of, for example, Sensorband, I tend to eschew large gestures, preferring instead to control musical processes with more subtle multi-touch gestures. My subtle gestures controlled algorithmic music for our performances, making me a conductor telling a laptop what musical directions to take. Almost all Aspect performances had a strong improvisational element, including my control over the algorithmic parts of our music. Like the other groups discussed in this section, Aspect experiments both technologically, with various kinds of computer inputs and animated scores, and musically via improvisation.

2.3 Summary

By placing computers as instruments into a larger context of advancing music technology and by providing an overview of ensembles that experiment with both music and technology, I have covered the musical context for my research. This sets the stage for the next chapter which includes background from the perspective of multi-touch instruments, networked instruments, and software toolkits.

Chapter 3

Related Work

The Hub has always been a collective of technically savvy musicians; we are all aware that one must maintain a very difficult balance between technology and expression. The trick has always been to get the tools working and then to find the music in the newly built context.

–**Scot Gresham-Lancaster** [41]

In this chapter, I describe related work that is relevant to my research in three areas: multi-touch instruments, networked instruments, and software toolkits. Accordingly, the chapter is broken into three sections. The first describes developments in multi-touch interactions both generally and in a musical context (Section 3.1). The second describes significant performances in the history of networked music and includes descriptions of instruments specifically designed for networked music (Section 3.2). The last section describes software toolkits for building multi-touch interfaces and for mapping human input to musical control (Section 3.3).

As new ways to interact with computers have developed, those interactions are invariably used in a musical context. Early computers used punched cards as a way to write programs. With the advent of the home computer, the keyboard and the mouse became the primary ways to interact with computers. As the home computer was becoming more common, networking was becoming easier. This allowed people to interact with computers that were not in the same room with them. Eventually, the mouse and keyboard style of interaction gained competition from the development of multi-touch interfaces in which people could directly interact with the user interface using touch. During all of

these developments and changes in interaction, software libraries and toolkits have been an essential part of allowing people to use the new interaction paradigms.

The research presented in this thesis builds on related work in three major areas: the development of multi-touch as a significant form of human-computer interaction, networked music as a new way for musicians to interact with each other and with their computers, and the design and implementation of software toolkits for human input and for mapping that input to musical control. I drew upon each of these three areas in my research into developing a model for combining multi-touch and networking into a software toolkit. The idea behind combining these areas is to achieve Gresham's "balance between technology and expression" as stated in the opening to this chapter. By careful investigation of the technologies involved, their history, and their musical context, the balance can be tipped toward expression.

3.1 Multi-touch Instruments

In this section, I will describe some milestones in the history of the development of multi-touch as a human-computer interaction. It is not meant to be a thorough survey of multi-touch but rather to show some of the important steps that brought multi-touch interactions to the mainstream.

3.1.1 General Developments

One of the earliest multi-touch devices was a tablet created by Lee et al [63] in 1985. This tablet had a touch surface that tracked multiple touches based on capacitive sensing. When a person touched some part of the tablet, the touch triggered a change in the capacitance of sensors beneath the tablet's surface, allowing for both touch detection as well as touch position sensing. This mode of touch tracking is used by current phone and tablet devices and so, as a prototype, it is a direct antecedent for the kind of interac-

tivity enabled by these devices. However, unlike current devices, this prototype had no graphical interface built into the tablet itself.

Another relatively early development in multi-touch was the DiamondTouch, a touch table created at the Mitsubishi Electric Research Laboratories (MERL) in 2001. The DiamondTouch [32] used capacitive sensing for tracking touches on the surface. More importantly, the DiamondTouch featured a graphical user interface (GUI) that was projected onto the surface of the table that responded to touches on the table. This combination of touch and GUI allowed for direct manipulation of on-screen widgets, providing a template for the touch-based systems that followed.

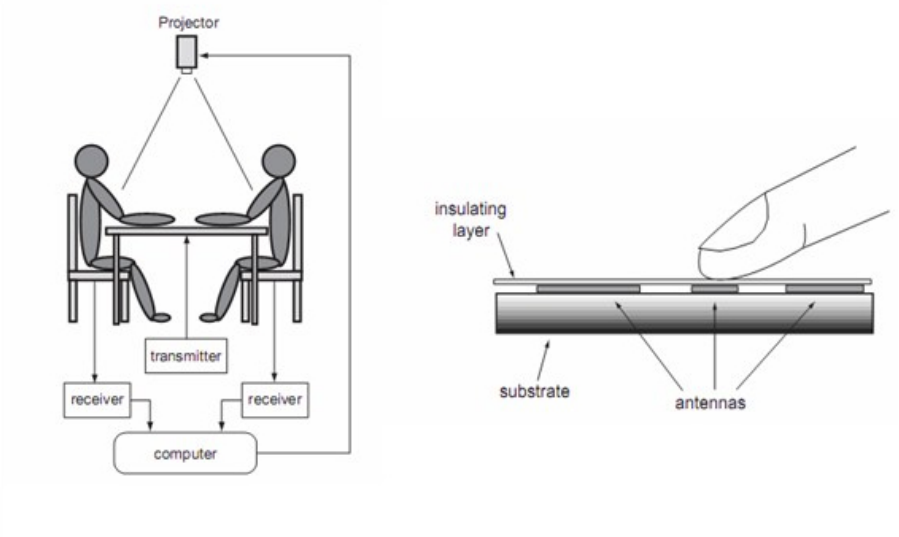


Figure 3.1: The design of the DiamondTouch tabletop system [32].

An important step in the evolution of multi-touch interfaces came with Han's introduction of the Frustrated Total Internal Reflection (FTIR) technique [45]. The touch sensing was done by an inexpensive web camera rather than by capacitive sensing as with the DiamonTouch. The camera input was filtered so that it looked in the infrared (IR) part of the light spectrum avoiding interference from light in the visible part of the spectrum. Since the edges of the clear plastic touch surface have infrared lights that shine through the plastic, any touch on the surface of the plastic will change the IR signature

near the touch, registering on the camera. Figure 3.2 is a diagram of the FTIR technique.

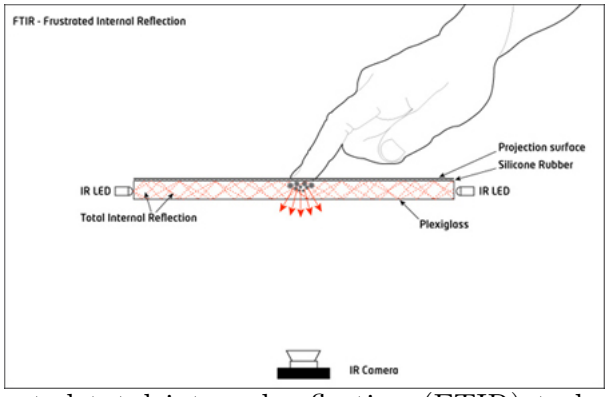


Figure 3.2: The frustrated total internal reflection (FTIR) technique [78] for tabletop touch tracking.

A projector under the interaction surface throws the graphical output on the underside, allowing for real-time interaction with the graphics without blocking the projector as happened with the DiamondTouch. FTIR tables were relatively cheap and easy to build, allowing multi-touch to reach outside of the laboratory and become a mainstream interaction technique.

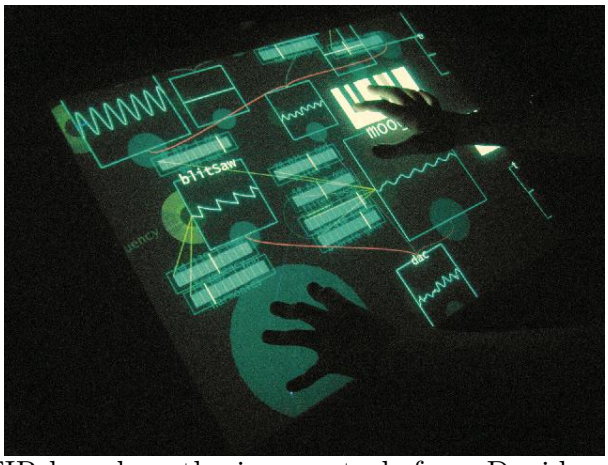


Figure 3.3: FTIR-based synthesizer controls from Davidson and Han [27].

Davidson and Han [27] used Han's FTIR technique to create a synthesizer control interface, showing that FTIR tables can be used in a musical context. The synthesizer

interface offered a variety of widgets borrowed from hardware interfaces, including sliders, knobs, and keys.

Multi-touch truly hit the mainstream with the introduction of the Apple iPhone in 2007. The iPhone, running the iOS operating system, used a capacitance-based system under the display hardware to detect touches, allowing for direct manipulation of the GUI. However, unlike earlier devices, the iPhone was quite small and was easily available to anyone who could afford it. After the introduction of the iPhone, Google offered its Android operating system on a variety of devices similar to the iPhone in terms of multi-touch capability. Eventually, Apple offered the iPad, a larger multi-touch device that is not a phone with additional features but is a general purpose computing system. Similar Android devices from a variety of manufacturers soon followed. All of these devices offer a huge number of music-making apps.

3.1.2 Musical Instruments

Multi-touch musical instruments have some antecedents that are not multi-touch but are still significant in the development of multi-touch instruments since they serve as models for later work.

The Jam-o-Drum by Blaine and Perkis [5] is not multi-touch but it is a sonigraphical instrument. It used commercially available drum pads for input. It did feature a top-projected visual interface, similar to the setup used for the DiamondTouch, showing that top-projected tables could be used in a musical context. Jam-o-World [4] expanded work on the Jam-o-Drum so that each drum pad was mounted on a turntable, adding an additional musical interaction. The Jam-o-World project is interesting in that it allowed for multiple players to effectively play the same instrument.

The Audiopad [85] allowed a player to control music by placing pucks on a tabletop

Figure 3.4: The Jam-o-Drum in use [3]. Note how the interface is projected onto the outstretched arm of one of the players.

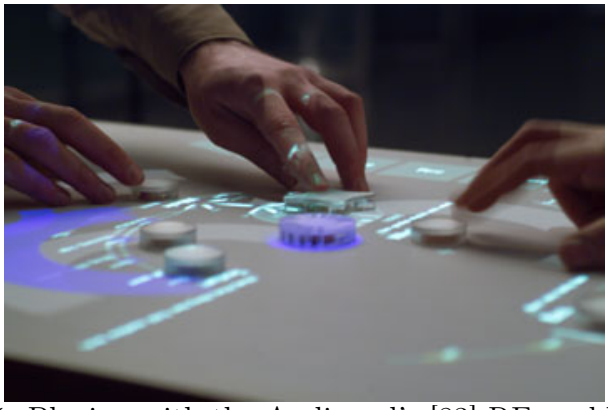


Figure 3.5: Playing with the Audiopad's [83] RF-enabled pucks.

surface as shown in Figure 3.5. Audiopad continued the development of the Sensetable [84] which used a very similar interaction system involving pucks that can be identified by radio-frequency (RF) tags in the pucks. The pucks represented various unit generators like sample audio file players and filters with each kind of puck identified by an RF tag. The parameters of each unit generator could be changed by manipulating the puck, either rotating or translating it. Like the Jam-o-Drum, the interface is projected onto the top of the tabletop surface.

The reacTable [56] was an important milestone in the development of multi-touch instruments. The reacTable takes input from multi-touch and from fiducial markers. The markers are a set of discs with a pattern printed on them that can be recognized

Figure 3.6: The reacTable with fiducial markers in play.

by a video camera under the table. The reacTable interaction via fiducial discs is quite similar to the Audiopad puck interaction but since it is based on camera tracking, it allows for multi-touch as well.

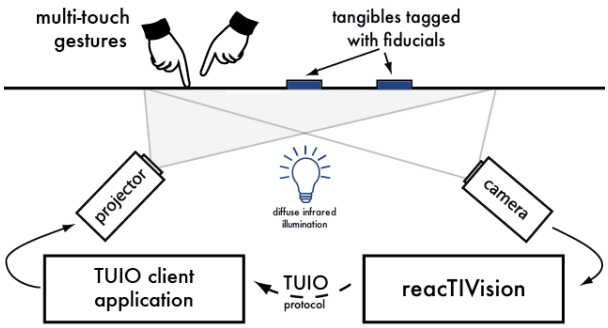


Figure 3.7: The setup [92] of the reacTable touch tracking system.

The camera tracking system, reacTIVision [57], is similar to how touches are recognized in Han's FTIR system described earlier but instead of putting IR light into a plastic touch surface, the IR light is projected in a diffuse manner from below the surface. The technique, diffused illumination (DI), is diagrammed in Figure 3.8.

Both fiducial markers and touch input on the reacTable are encoded in TUIO messages [58], an OSC namespace that allows any camera-based system to send touch tracking data like the number of touches and their locations to software that understands TUIO.

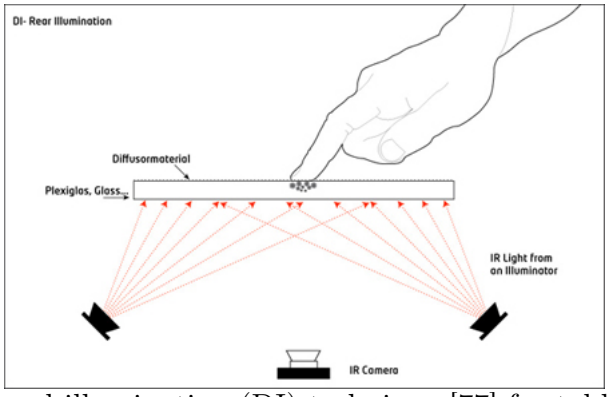


Figure 3.8: The diffused illumination (DI) technique [77] for tabletop touch tracking.

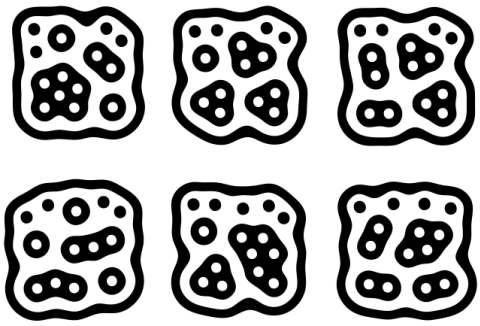


Figure 3.9: The fiducial marker patterns [92] used to control the reacTable.

The reacTable was not only one of the first musical instruments to have multi-touch input, it was also a highly configurable instrument based around the idea of Max Mathew's unit generators. Fiducial markers could represent oscillators, filters, sequencers, or sound file players among others. These could be combined in a variety of ways, similar to the design of the Audiopad system. This meant that the reacTable could be used to create many different kinds of instruments with a wide variety of sounds and interactions. The reacTable, now called the Reactable Live! is available as a commercial product [90] and eventually, the Reactable company released the Reactable Mobile [91], a software application for iPad and Android tablet devices. Rather than having the focus on fiducials as with the tabletop version, the Reactable Mobile uses multi-touch only.

The JazzMutant Lemur [52] was one of the first commercially available multi-touch devices specifically for music. It features a variety of configurable widgets like faders,

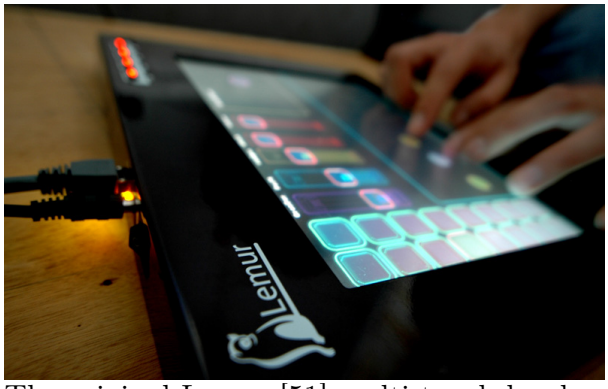


Figure 3.10: The original Lemur [51] multi-touch hardware controller.

knobs, and switches and each widget could be mapped to send either MIDI or OSC messages. This is in contrast to the previously discussed instruments that featured a direct mapping between user input and control of a particular audio engine. Instead, with the Lemur, generic input controls could be mapped [127] to virtually any kind of audio control. For example, a fader widget could be mapped, via OSC message, to the audio gain. As long as the audio engine controlled by the Lemur understood the messages sent from the interface, any mapping was possible. The Lemur has ceased production as hardware and is now available a multi-touch software application for iOS and Android devices [64]. The Lemur software retains its ability to connect, via networking, to a variety of audio engines.

3.2 Networked Instruments

This section will focus on musical performances and instruments that feature some form of computer networking along with some points of interest in the general development of computer networking. Networked music performance is a sizable area of research that can be broken down into two broad categories: 1) message-based network performances and 2) audio-based network performances. The focus of this section is on instruments and performances that are message-based since this is the most relevant to the research

presented in this thesis.

Weinberg [118] describes the history of what he calls "Interconnected Musical Networks" beginning with the early transistor radio experiments of John Cage. Weinberg considers Cage's *Imaginary Landscape No. 4* to be the first electronic "Interconnected Musical Network". While this is an interesting starting point to choose for networked music, I will start my examination at a later point than this. The Cage piece does not represent a true modern network in the sense that radios are one-way only and therefore do not represent a true live platform for music performance. In other words, radios do not have the give-and-take that is offered by modern computer networks in which musicians can interact with each other using networks as a medium.

One of the earliest attempts to create a musical networking system came in the form of the Musical Instrument Digital Interface (MIDI) specification [73], first published in 1983. MIDI was created to allow for the exchange of musical information such as pitch and timing to be sent over specialized hardware and cables. MIDI was initially created with digital keyboards [68] in mind but has since been adapted to a variety of digital instruments.

Around the time of the development of MIDI, in 1981, the specification [88] for the fourth version of the internet protocol, IPv4, was released. The IPv4 protocol is significant in that it became the protocol that powered the internet and eventually almost every kind of computer network. The protocol is independent of the underlying networking hardware, making networking between different computing systems easier. Together with the Transport Control Protocol [13] and the User Datagram Protocol [86], IPv4 enables almost all networking traffic.

The first serious messaging system for music to take advantage of networking over generalized computer networks (unlike MIDI) is Open Sound Control [126]. The OSC

Table 3.1: Example OSC Messages.

Address	Type Tag	Arguments
`/instrument/gain`	`f`	`0.5`
`/instrument/pitch`	`i`	`2`

specification [124], first published in 2002, provides for a content format that can be used in a variety of networking contexts, including over standard TCP- and UPD-based networks. Unlike MIDI, OSC is a completely open content format that allows developers to create arbitrary messages. A standard OSC message has three parts: an address string, a type tag string, and a series of arguments that matches the types in the type tag string. Table 3.1 shows example OSC messages separated into address, type tag, and arguments.

OSC does not specify anything about the message beyond a list of acceptable type tags (that can be found in the specification), making it extremely flexible for sharing different kinds of musical data between computers. As long as the messages are understood by all computers on the network, those computers can become part of a single musical system.

Latency [60] is invariably an issue for networked instruments. In an important study on the effects of latency, [14, 43], observers examined the effects of latency on ensemble accuracy by having pairs of participants attempt to clap in time with each other. The participants could only hear each other over headphones. During the study, an artificial delay was created between the participants to determine the effects of the delay on their ability to clap a rhythm with a specified timing. On examining the results, the experiment showed that any delay of more than 35 ms had a significant negative effect on the accuracy of the participants clapping. The results showed an accuracy sweet spot of approximately 11.5 ms with participants speeding up the tempo at values less than this and participants slowing down at values more than this.

In general, network latency is a matter of physics with the speed of light as the theoretical maximum. This being the case, Cáceres and Renaud describe [25] their work

in using latency as a musical device rather than by trying to solve the problem of latency. Brandt and Dannenberg [6] discuss the notion of timing, including latency and jitter, in real-time music performance systems. A full discussion of timing is beyond the scope of this thesis but is an important issue to mention in the context of networked music.

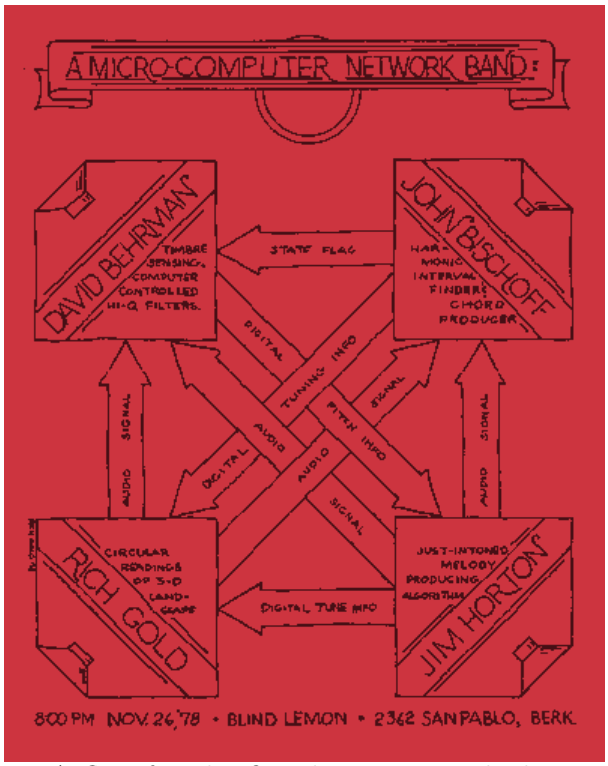


Figure 3.11: A flier for the first know networked music concert.

3.2.1 Performances

The League of Automatic Music Composers performed at the first known networked music concert in 1978 [2] at the Blind Lemon club in Berkeley, California. The group used KIM-1 microcomputers [122] connected with cables that enabled them to share musical information between their respective microcomputers. During this performance and others, the group used a variety of networking configurations. In one configuration, one of the performers would send the output of his computer to another performer's computer for manipulation before the final audio output. This performance showed the range of

new musical interactions afforded by computer networking.

Figure 3.12: The League of Automatic Music Composers in 1981.

The Hub [41] evolved from the League of Automatic Music Composers, continuing their experiments with networked music. In 1985, the Hub performed over a modem from two sites in New York City, one of the first multi-site internet-based performances. Eventually, around 1990, the Hub moved to using the MIDI protocol for their performances. The Hub were one of the early adopters of OSC, using it in an internet-based performance in 1997 to send MIDI data over the internet. Gresham-Lancaster reflects on that performance, in the full quote that begins this chapter with emphasis added:

> **This formidable test actually ended up being more of a technical exercise than a full-blown concert.** The Hub has always been a collective of technically savvy musicians; we are all aware that one must maintain a very difficult balance between technology and expression. The trick has always been to get the tools working and then to find the music in the newly built context. **In this case, the technology was so complex that we were unable to reach a satisfactory point of expressivity.**

This quote from a technically-experienced musician shows the difficulty of early real-time networked music performances at that time. The quote is also important in the

context of this thesis in that it shows the importance of providing tools that simplify the process of playing music over networks while at the same time allowing for creative flexibility in that same networking context.

3.2.2 Instruments

Networked musical instruments can be roughly broken down into two categories: those that exchange audio signals over networks and those that exchange control messages (usually via OSC or MIDI). The focus of this subsection is on instruments that exchange control messages over networks since that is the most relevant to the work presented in this thesis. Networked audio instruments have their own issues as discussed in Renaud et al [95], Barbosa [128], and Mills [74]. Weinberg [119] offers a survey of networking configurations (or topologies), and discusses centralized versus decentralized configurations.

As discussed earlier, latency is an issue with networked instruments since it takes time for musical data to be passed between players over a network. This is especially true when that network is the internet though it is less of an issues for local networks. Message-based instruments are still subject to the limitation of latency but the situation is not as acute as it is with audio-based networking since audio that is not delivered consistently and on time will produce very noticeable glitches. Messages that do not arrive on time or at all are still a problem but this may not even be noticeable depending on what those messages control. In other words, message-based systems have the potential to be more adaptive to networked situations.

In 1990, the NetJam project [61] started, allowing participants to submit MIDI data to a server via e-mail that would then send that data out to all other participants. NetJam was not a real-time system but it is a good example of making use of existing technology, in this case e-mail, for the purpose of making music in a collaborative fashion. Essentially, e-mail acted as a messaging system for musical information.

Faust Music Online (FMOL) [53] was a relatively early system for collaborative performance and composition over the internet. Performers and composers would download client software and the clients would then share small proprietary score files based on input from performers. Those score files would then be sent to a central server that kept track of the score files for each client. The score files can then be downloaded from the server to drive the audio engine built into each client. Clients can see and share each other's score files as well as manipulate any score file and send it back to the server.

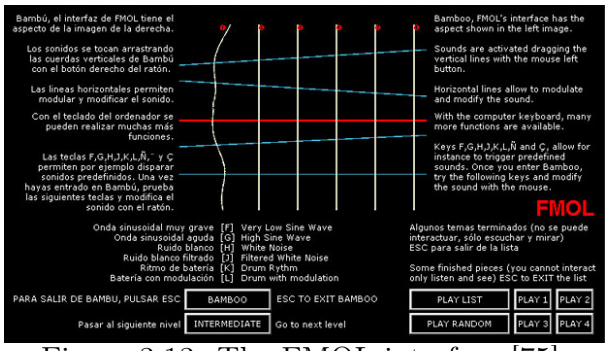


Figure 3.13: The FMOL interface [75].

Quintet.net [44] used modified MIDI messages that are transmitted from each client node to a central server. The MIDI notes are then sent from the server out to performers as well as to audience members. This system is notable in that there is another role aside from performers and audience members, that of a conductor. The conductor in this system has control over both the sounds and the effects used by clients in addition to being able to send out short scores to performers.

JamSpace [42], created by Michael Gurevich, was a message-based, client/server music system. Using this system, the performers could create short phrases using a custom-made JamPad interface made of 12 pressure-sensitive pads. Performers recorded their phrases and played them back in their own client interface. The JamSpace client interface is shown in Figure 3.15. The recorded phrases could then be added to a public JamSpace server where any other client on the network could download the phrase and play it back.

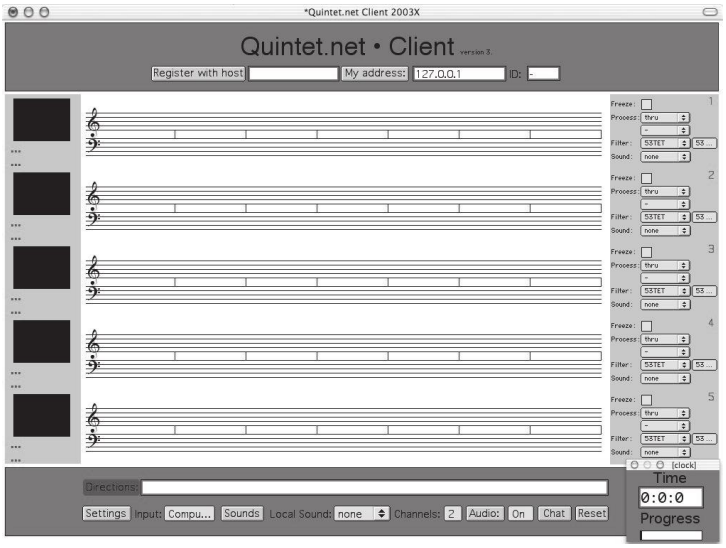


Figure 3.14: The Quintet.net client interface.

JamSpace showed the viability of the client/server networking in sharing and combining musical ideas. As Gurevich points out, the JamSpace system worked well for real-time sharing because it operated on a local network rather than across the internet, minimizing network latency.

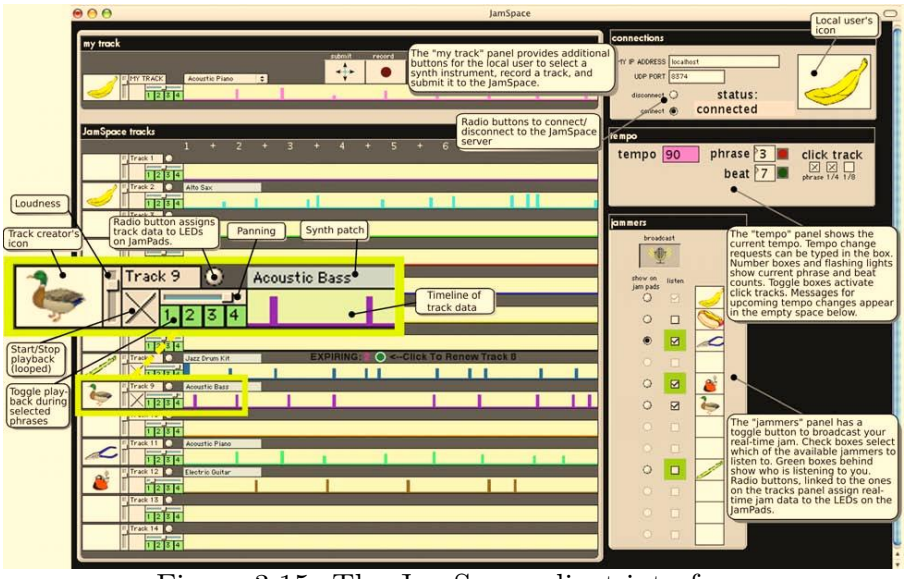


Figure 3.15: The JamSpace client interface.

The internet served as a the medium for the web-based Public Sound Objects [129], created by Barbosa and Kaltenbrunner. A server handles all sound synthesis based on

input from clients. The synthesized sound was then sent back to the client via streaming audio. The control interface features a moving ball that bounces off of the "walls" of a square, triggering a sound each time the ball touches the wall. Performers can control various parameters of the ball including size, speed, and direction. Figure 3.16 shows the control interface with a bouncing ball.

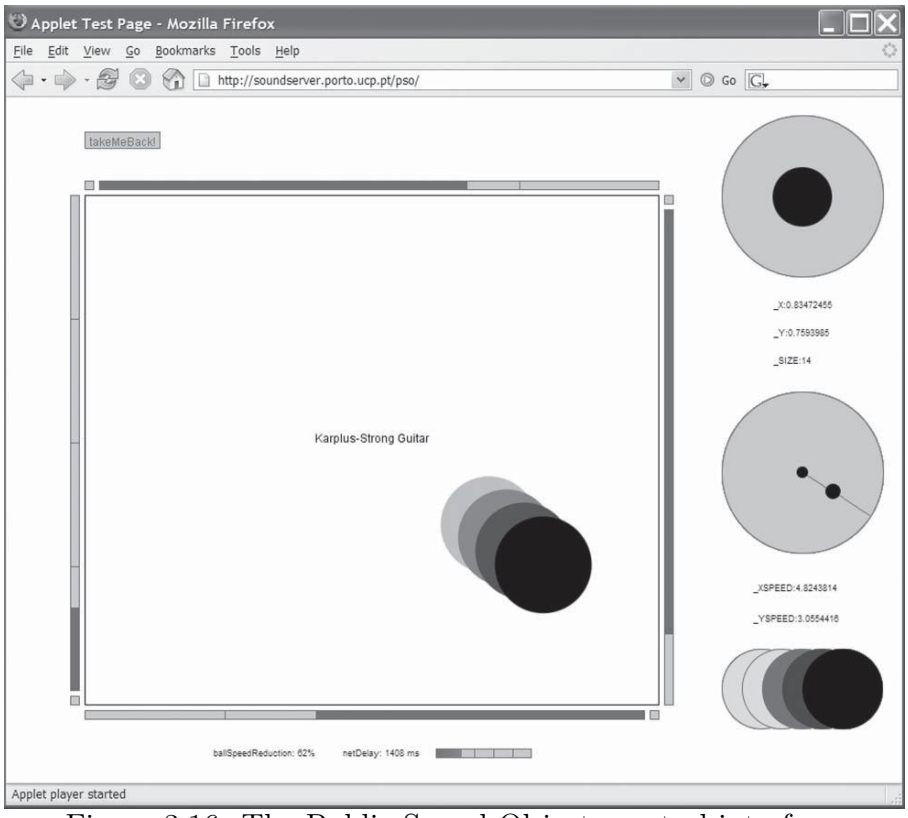


Figure 3.16: The Public Sound Objects control interface.

A message-based system, peerSynth [105] was designed and implemented by Stelkens using peer-to-peer (P2P) communication enabling clients to connect directly with no server involved. The author makes the point that peerSynth saves bandwidth by using messages with synthesizer control parameters rather than audio between clients. Interestingly, the latency between clients can be used to control parameters of the synthesizer as well, making peerSynth an instrument that not only accounts for latency but actually makes use of it in a musical context.

The idea of using latency as a musical device is also present in the work of the Net vs. Net collective [25]. The Frequenciliator [94] by Rebelo and Renaud combines audio and message-based networking. Each performer sends an audio stream to the other performers. Timing, however, is controlled by messages. This includes countdowns to upcoming events (to give performers some warning) and general synchronized events that can be sent to any or all performers and then mapped locally by the receiving performer.

3.3 Software Toolkits

Software toolkits are an essential element of building interfaces, musical or otherwise. At this stage in the evolution of computing, it is almost impossible to build a human interface that does not build upon some previously developed software. Almost all software is part of a stack with the developer's software resting on top of other software that is itself built on top of software and so on. The major point with regard to this thesis is that toolkits (libraries with a more action-oriented name) are an absolutely essential part of building modern software interfaces.

The toolkits described below are related to the research presented in this thesis in that they provide some kind of functionality that overlaps with my research. Toolkits are broken down into multi-touch and mapping toolkits. The multi-touch toolkits discussed take in touch input and provide output for graphics. The mapping toolkits take touch input and provide output for graphics and for musical control messages.

3.3.1 Multi-touch

The DiamondTouch toolkit [29] was an early collection of functions for supporting multi-touch input on the DiamondTouch system. MT4j [62] (Multitouch for Java) is a multi-touch toolkit that allows for a wide range of multi-touch input with integrated graphics. SMT is the Simple Multi-touch Toolkit [81] works much like MT4j but without built-

in graphics. SMT is a library for the Processing graphical programming environment. Another Processing library is tuioZones [69]. Both SMT and tuioZones support TUIO and mouse input. PyMT [46] is similar to the previously mentioned toolkits but is written in the Python programming language. Boing [80] is yet another toolkit that is written in Python. LightTracker [38] is an all-in-one solution for vision-tracking systems that includes components for both tracking touches in hardware and software. Other toolkits are mentioned in Kammer et al [59] along with a taxonomy to categorize them. None of the previously-mentioned toolkits supports any kind of mapping to messages, either OSC or MIDI. They only support multi-touch with graphical output. This means that these toolkits are related only to the multi-touch aspects of my research, not addressing the mapping part of it.

3.3.2 Mapping

In the context of music performance systems, mapping is the process of taking human input of some kind and then mapping that to control of musical parameters. Human input can involve almost anything from multi-touch to waving hands in front of a camera to just about any connection that can be made between human action and some kind of musical control. Ultimately, any form of human input will be translated into numbers and those numbers can then be mapped to musical parameters. For a simple example, a touch on a tablet interface could be mapped to the control of a synthesizer. By moving a touch in the X direction, a person could change the pitch like moving across the keys of a piano but with the option of continuous changes of pitch instead of in discreet notes. Also unlike a piano, movement in the Y direction could control the volume of the notes played with continuous changes as a touch moves up and down the interface. This is the mapping of interaction or gestures to musical control. That mapping can occur in a variety of ways, but the most flexible is mapping via OSC [127] messages.

Control [99, 100] is a toolkit for building widgets like buttons, sliders, knobs and multi-touch XY pads that can be mapped to OSC or MIDI messages. The widgets are created using JSON code [7] with the following code creating a button.

```
{
    "name" : "myButton",
    "type" : "Button",
    "x" : 0, "y" : 0,
    "width" : .25, "height" : .25,
}
```

Figure 3.17 shows an example Control interface with buttons, sliders, and knobs. The features of Control are described in more detail in Chapter 5, Section 5.1.

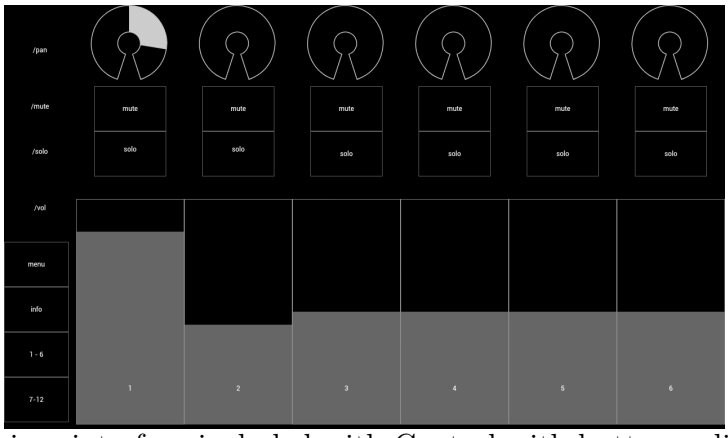


Figure 3.17: A mixer interface included with Control with buttons, sliders, and knobs.

TouchOSC [47] offers a set of widgets that is similar to the ones available in Control Like Control, those widgets can be mapped to arbitrary OSC or MIDI messages. Available interactive widgets include push buttons, toggle buttons, XY pads, faders, encoders, and sets containing multiples of all of these (other than encoders). Developers use a specialized editing program called TouchOSC Editor to create layouts featuring one or more widgets. That layout can then be loaded onto a multi-touch device. Once loaded onto the device, the interface can be configured to connect to a host that receives OSC

or MIDI messages.

Figure 3.18: An interface built with TouchOSC [47].

As mentioned earlier, in the section on multi-touch instruments, the Lemur hardware controller is no longer in production and has been replaced by the Lemur app. That app features a customizable set of widgets like buttons, faders, knobs, and pads. These are similar to the widgets available in the TouchOSC and Control. As with these other toolkits, the Lemur app allows widgets to map to MIDI or OSC. Like TouchOSC's editor, widgets are loaded onto a device using a specialized application. The appearance of widgets can be changed by the application of custom templates. The Lemur is notable in that it features a physics engine in the interface that allows interface elements to bounce, rebound, or oscillate.

Mira [22, 110] offers a set of widgets for multi-touch devices that is designed specifically to connect to the Max [24] audio programming environment. Mira's widget set includes buttons, toggles, dials, and sliders among others. These widgets are mirrors of widgets built in the Max interface that Mira connects to. While Mira is a useful way to build controllers for use with Max/MSP, it is less useful for musicians who use other programming environments. Figure 3.20 shows a Mira controller. Notice how the controller replicates the Max interface running on the laptop.

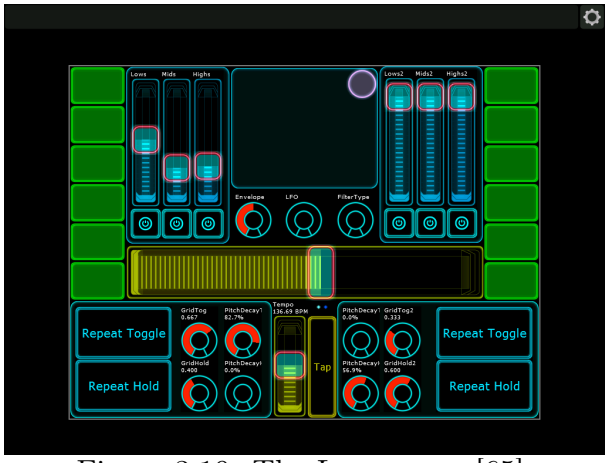


Figure 3.19: The Lemur app [65].

Figure 3.20: A Mira controlling a laptop running Max/MSP [22].

TouchOSC, Control, and Lemur are quite similar to JunctionBox in terms of functionality and interactions. However, there are significant differences in philosophy between these three and JunctionBox. TouchOSC, Control, and Lemur are designed to make it quick and easy to build multi-touch interfaces using well-known widgets with a set visual design. JunctionBox is designed to allow for a much greater range of both multi-touch interactions and combinations of interactions in tandem with few constraints on the visual design of interfaces. TouchOSC, Control, Lemur, and JunctionBox are compared in greater detail in Chapter 5.

3.4 Summary

By describing research in multi-touch instruments, networked instruments, and software toolkits, I have illustrated how computer music has rapidly incorporated new technologies. This practice of making the best use out of available technologies has shown the demand for access to multi-touch and networking capabilities as well as the importance of having toolkits that address those capabilities.

Chapter 4

JunctionBox

I am a strong believer that tools shape the work of researchers, artists, and even product developers, so we should pay close attention to the tools that exist (they determine a lot about short-term progress), and tools that are needed (the need slows progress and diverts efforts into less productive directions.) Tools in this domain are generally software applications, but we also need to consider libraries, plug-ins, frameworks to contain plug-ins, protocols, and languages.

– **Roger Dannenberg** [26]

In this chapter, I describe my development of a *unit interaction* model for multi-touch interactions and its reification via my JunctionBox creative coding toolkit. The research question that I address in this thesis is whether Max Mathews' *universal building blocks* concept can be applied to multi-touch, networked interactions, leading to a *unit interaction* model. To answer this question, I designed and built the JunctionBox interaction toolkit as a reification of the *unit interaction* model that I developed. The goal of creating the *unit interaction* model was to parallel the concept of universal building blocks by identifying the most fundamental mappable multi-touch interactions. The process of creating JunctionBox both instantiated the unit interaction model in usable form and served as a vehicle for further development of the *unit interaction* model as the toolkit was being built. In other words, creating JunctionBox itself served to help identify the most fundamental mappable interactions. This chapter describes the mappable interactions, implemented in JunctionBox, that serve as the basis for my *unit interaction* model.

The chapter is divided into five sections. After defining unit interactions (4.1), the next section (4.2) describes the fundamentals of JunctionBox design and functionality. The third section (4.3) contains a detailed description of each unit mappable multi-touch interaction offered by JunctionBox along with a diagram describing that interaction. The next section (4.4) details the special features in JunctionBox that go beyond interaction mapping. The last section (4.5) provides a short summary.

4.1 Defining Unit Interaction

To begin to develop a *unit interaction* model based on the concept of universal building blocks, I first defined unit interactions in the context of multi-touch interactions that map to musical control. Unit interactions in this context must meet the following requirements for inclusion in the model:

1. Each interaction must be the most fundamental action that can be performed with one or more touches on a two-dimensional surface (e.g. moving a widget across the screen in the X and Y directions) or a gesture commonly used on multi-touch devices (e.g. 2-touches to scale the width and height of a widget).

2. Each interaction must be individually mappable to musical control.

3. Each interaction must be networked.

4. Each interaction must have output for graphics.

The interaction building blocks that meet these requirements are the *unit interactions* that make up the model. The following section on fundamentals describes how basic functionality is achieved. This is followed by a section that explains how the unit interaction model is integrated with the basic functionality.

4.2 Fundamentals

The name JunctionBox is inspired by the following definition of a junction:

> **junction**: [**juhngk**-shuhn] a place or point where two or more things meet or converge. [30]

For JunctionBox, the convergence is among multi-touch input, graphical output for visual feedback, and musical control via message mapping. The meaning of the word junction highlights the importance of mapping to the design of JunctionBox. The box in JunctionBox is a reference to the rectangular nature of (nearly all) multi-touch devices.

JunctionBox is as a library for the Processing creative coding environment [111]. Processing is primarily geared toward creating graphics but it has support for libraries that augment its core capabilities. As a library, JunctionBox augments Processing graphics by taking multi-touch input and mapping that input to musical control. JunctionBox takes multi-touch input and maps it to two outputs: 1) message output and 2) output for graphics. Messages are destined for some kind of audio engine and graphics appear on the screen of the device. Figure 4.1 shows JunctionBox as a hub for mapping touch input to message and graphical output.

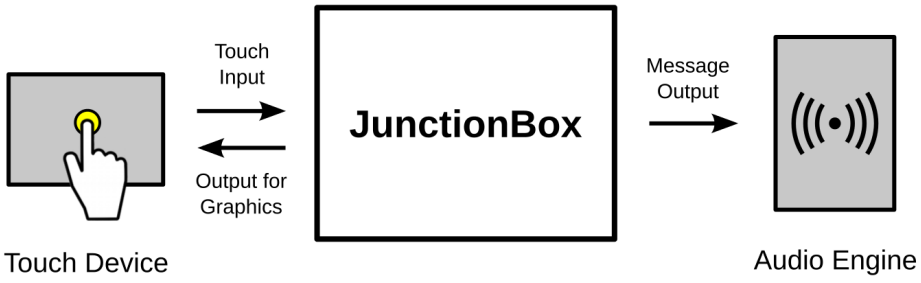


Figure 4.1: JunctionBox takes touch input and maps it to message output for controlling an audio engine for music and returns output to the device for controlling graphics.

JunctionBox outputs values to the Processing [93] graphics engine. JunctionBox enables the creation of what Jordà calls sonigraphical instruments [54]. These instruments feature both sonic and graphical output. Processing works well because it offers a very capable graphics engine, allowing work on JunctionBox to focus on supporting interactions, especially in a musical context. JunctionBox bridges the interaction gap between the graphical possibilities offered by Processing and the sound control possibilities afforded by the Open Sound Control (OSC) protocol [127].

4.2.1 Multi-touch Input

To map multi-touch input to output, JunctionBox takes in basic touch data including identifiers for each touch and the X,Y location for each touch on a device's screen. The identifier (ID) allows touches to be distinguished from each other while the X, Y location is used for mapping to output. Figure 4.2 shows the basic touch input used by JunctionBox.

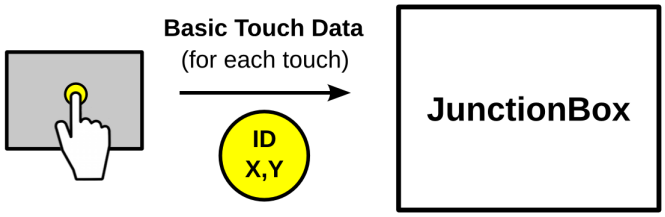


Figure 4.2: JunctionBox takes basic touch data as input.

Basic touch data comes to JunctionBox in one of two ways: 1) from TUIO [58] messages or 2) from Android [35] touch tracking. In either case, the basic touch data is the same. TUIO is used mainly in larger tables (like the reacTable) while Android is generally used for smaller phone and tablet devices. JunctionBox can also handle mouse input with the mouse cursor acting like a single touch point.

Once touch data is received by JunctionBox, it is dispatched to the interactive parts of the interface, called Junctions. A Junction is defined by its shape, location, and area.

An interface can have multiple Junctions with each one having either a rectangular or an elliptical shape. If one of more touches occurs inside the location and area of a Junction, that touch data will be sent to the Junction, initiating one of the interactions described in Section 4.3. Touches that occur outside of any Junction have no effect. Figure 4.3 shows a rectangular Junction that turns green when a touch occurs inside of its area.

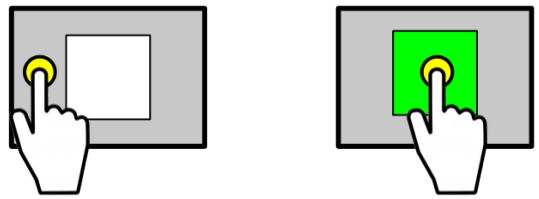


Figure 4.3: Touches that occur outside of Junctions (1) have no effect while touches that occur inside (2) initiate an interaction.

The shape of a Junction is important in determining whether a touch is dispatched to that Junction since rectangles and ellipses require different approaches to figuring out whether a touch occurs within their respective areas. In order to accomplish this, Junctions check their shape to determine which approach to take. For rectangles, this is a check of the location of the center and the area. For ellipses, it is the center and the distance from the center. Rectangle and ellipse do not need to be squares and circles to work properly. Rectangles work the same way that squares work but ellipses with axes of different sizes require some trigonometric calculations for determining whether touches are inside. Junctions handles this internally by checking for different sized axes. These checks ensure that interface builders never need to worry about the shape of a Junction. Touch data will always be handled correctly regardless of whether rectangles or ellipses have different-sized axes.

4.2.2 Message Output

All interactions with Junctions are mappable to messages that are sent to an audio engine. More specifically, interactions with Junctions are mapped to OSC messages. As described in Chapter 3, OSC is an open message specification used in a large number of music performance systems. Part of the appeal of OSC is its flexibility [125] in adapting to a variety of musical situations. The flexibility also applies to mapping as discussed by Wright et al [127], making OSC an obvious choice for mapping in JunctionBox. Audio engines that support OSC include ChucK [117], Max/MSP [23], PureData [18], and SuperCollider [28]. Each of these audio engines offers a programmable environment for generating music. In using any of these engines, certain musical parameters are set within the engine and the messages will control those parameters. The following is an example message that could control the gain (volume) of an instrument built with an audio engine.

```
/instrument/gain 0.5
```

The first part of the message is called the address (/instrument/gain) and that address can be almost any combination of letters and numbers. Parts of an address can be separated by a "/" in cases where there might be multiple parameters (gain, frequency, etc.) for the same instrument. As long as the Junction controlling the engine uses the same message as the engine itself, the Junction will control that parameter with the value set in the message (0.5). This is the essence of using OSC for mapping in general and for JunctionBox in particular.

4.2.3 Networking Options

JunctionBox uses computer networking to connect to audio engines. With computer networking, there are a range of options for where the audio engine is relative to the JunctionBox interface. They can both be on the same computer, they can be on different

computers but in the same room, or they can be on different computers connected over the internet. OSC messages can be used in any of these networking situations.

JunctionBox supports OSC messages delivered with the User Datagram Protocol (UDP) [86] on any internet protocol (IP) [88] network. All that is needed to connect JunctionBox to an audio engine is an IP address and a port number. This is analogous to delivering a package to an apartment where you would need both the street address and the apartment number in order to get the package to the correct apartment. In JunctionBox, the address and the port can be specified for an entire interface or per Junction. This means that individual controls on an interface can map their interactions to different audio engines.

4.2.4 Output for Graphics

Junctions create output for graphics in addition to mapping messages, allowing Junctions to have a defined appearance on the screen of a multi-touch device and to change that appearance based on feedback from interactions. All graphical output is handled by Processing. When a rectangle or an ellipse is drawn in Processing, it will take data from a Junction including location, angle, and size. All of these can change by interacting with a Junction and that change will be reflected graphically. Shape aside, the graphics associated with a Junction are not pre-defined. The combination of Processing graphics and JunctionBox interaction mapping allows for a great deal of freedom in interface design.

In JunctionBox, OSC messages are sent as soon as an interaction occurs in a way that is entirely independent of graphical output. Using Processing for graphics means that visual changes occur at a particular frame rate. If messages were sent only when a frame is updated, then message sending would be dependent on frame rate. But more importantly, having messages sent only when a frame is drawn would introduce latency

or a delay in the sending of an OSC message. At Processing's default frame rate of 60 frames/second, the time between frames is 16.7 milliseconds, a significant delay for musical instruments according to Wessel and Wright [121]. Graphical output and OSC message output are decoupled in JunctionBox in order to avoid these delays.

4.3 Mappable Interactions

The mappable interactions described in this section constitute the *unit interaction* model and its implementation via JuntionBox. The following subsections discuss specific mappable interactions or categories of interactions. Activation and toggling are interactions. The remainder of the the subsections (translation, rotation, scaling, and touches) are categories of interactions. The diagrams in each subsection show how interactions affect Junctions. As such, they show a Junction as it changes from one state to another based on the described interaction. The different states of the Junction are separated by arrows in diagrams where this makes state changes more clear. In other diagrams, arrows represent state changes in a single image. For simplicity of representation, most diagrams show only a single touch. JunctionBox supports as many touches per Junction as the underlying hardware will permit.

4.3.1 Activation

The most basic mapping in a Junction is what I call activation where activation is defined as the presence or absence of touches within a Junction's area. For this interaction, a message is sent to indicate an active state as soon as the first touch is received. When the last touch is removed from a Junction's area, a message is sent indicating an inactive state.

Besides message mapping, Junctions can also provide output on their activation state for graphics, allowing for visual indicators (like a change of color) in an interface. Fig-

ure 4.4 shows a color change based on activation.

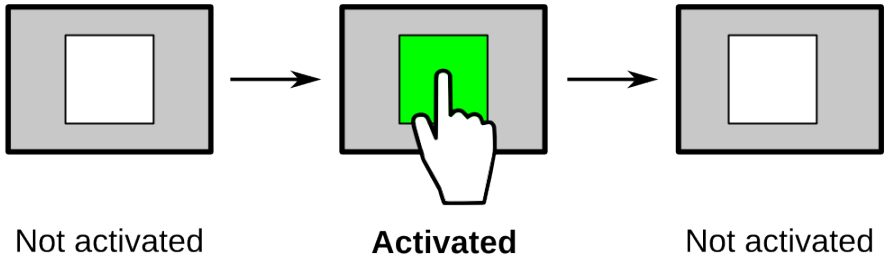


Figure 4.4: Activating a Junction with a single touch. When the touch is removed, the Junction is no longer active.

4.3.2 Toggling

A Junction has a toggle state that can be changed with a touch. The first touch sets the state to toggled and a follow-up touch will de-toggle the Junction. In Figure 4.5, the Junction turns green when it is toggled. The follow-up touch would de-toggle the Junction, turning it white again.

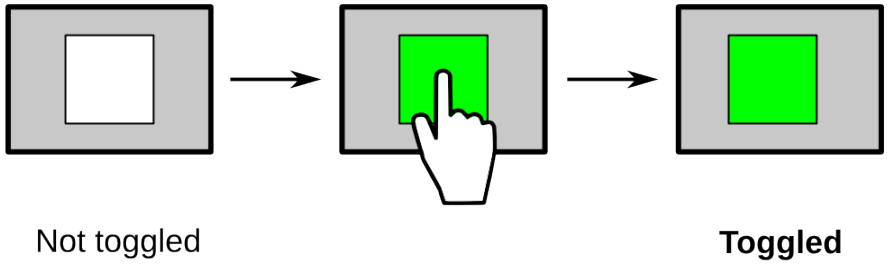


Figure 4.5: Toggling a Junction. De-toggling involves a follow-up touch when the Junction is toggled.

4.3.3 Translation

A Junction can be translated (moved) to any position on an interface. The X or Y values for the center of the Junction can be mapped to messages as it is translated. The mapping for this interaction sends both the X and Y values for the center or X and Y values separately. As with all Junction mapping, values for the center are normalized from 0 - 1 based on the width and height of the interface. Figure 4.6 shows a translated Junction

and the X and Y value ranges. Messages are only sent when the center of a Junction changes position. Translation limits can be set that restrict the movement between two points on an interface. When limit points are set, the values are still normalized based on those two points.

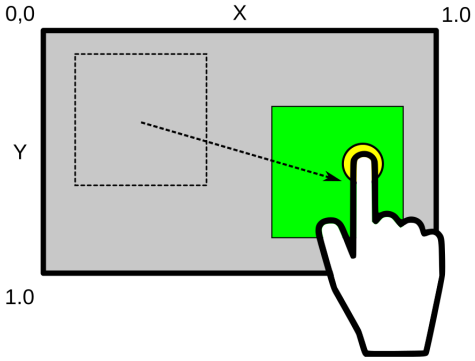
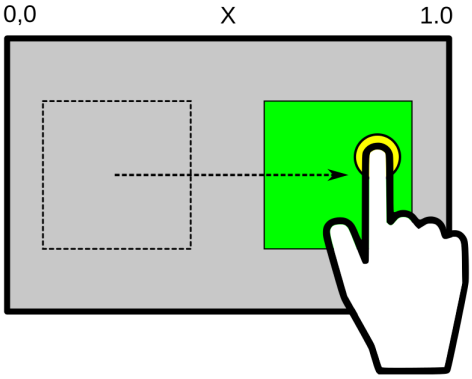
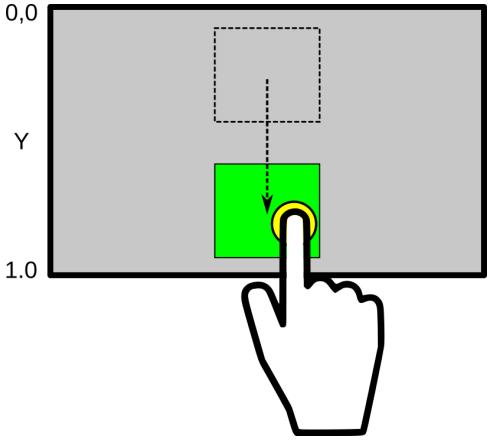


Figure 4.6: Translating a Junction will send a message with the X and Y values for the center of the Junction, normalized from 0-1.

Figure 4.7: Mapping Junction translation in the X or Y direction only.



(a) Translating a Junction in the X direction.



(b) Translating a Junction in the Y direction.

In addition to sending both values, a Junction can send only the X or Y value. These X or Y options are useful since Junctions can be limited to translation in either the X (horizontal) or Y (vertical) direction only. These X- and Y-only options also allow for different messages to be sent for both components of translation when both X and Y

translation are enabled.

The number of touches that translate a Junction is one by default but this number can be set to any number greater than one. Translation can also be enabled with a range of touches. For example, translation could happen with 1-4 touches. Regardless of the number of touches, the same center X and/or Y values for the Junction are sent as messages. Figure 4.8 shows a 4-touch translation.

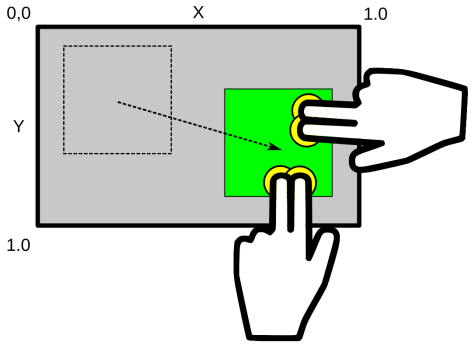


Figure 4.8: Translation can be enabled with multiple touch. In this case, four touches translates the Junction.

4.3.4 Rotation

Junctions can be rotated to any angle and the value of the angle can be mapped to messages. The angle is mapped starting with the 12 o'clock position. Before rotation, the 12 o'clock position represents a zero angle. Once rotated, the angle increases in the clockwise direction until the Junction nears its maximum 360 degree rotation angle. These values are normalized as shown in Figure 4.9 with a vertical line showing the 12 o'clock position. Once the Junction has passed the maximum 360 degree angle, the angle reverts to zero and increases again in the clockwise direction. This is repeated for each full rotation. Rotations in the counter-clockwise direction have negative values for the angle.

There are two kinds of mappable rotation interactions: a one-touch rotation and a two-touch rotation. One and two-touch rotations can be enabled or disabled indepen-

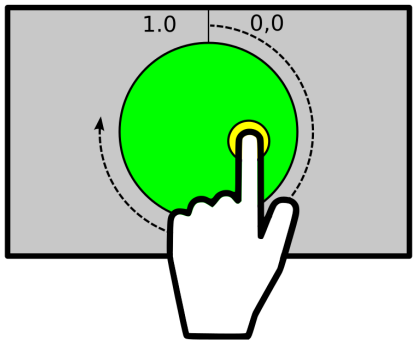
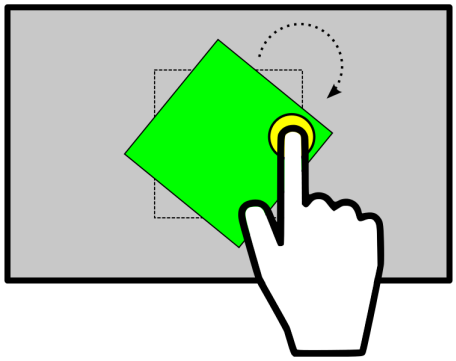


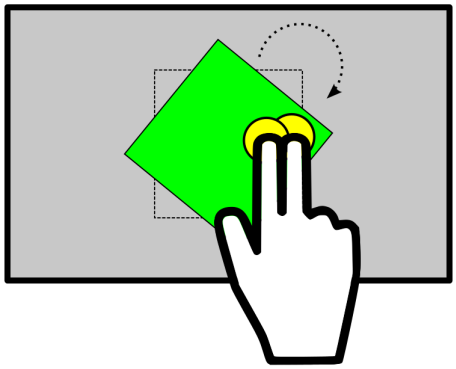
Figure 4.9: The rotation angle of a Junction starts at the 12 o'clock position and is normalized. The rotation shown is a one-touch rotation.

dently by specifying the number of touches. If the number of touches is not specified, then both kinds are enabled.

Figure 4.10: Rotating Junctions with either 1 or 2 touches.



(a) Rotating a Junction with one touch.



(b) Rotating a Junction with two touches.

The separation of interactions between one and two-touch rotations allows for a greater variety of mapping situations. If both kinds of rotation were enabled, each kind could be mapped to a different message, distinguishing between the two kinds of interaction using the same Junction.

Junctions keep track of the number of rotations internally. Each full rotation in the clockwise direction increases the rotation count. Rotations in the counter-clockwise rotation will roll back the number of rotations. The number of rotations can be mapped

to messages.

4.3.5 Scaling

The width and height of Junctions can be changed by scaling with two touches. The values for width and height (or both) are then mapped to messages. The change in scale is proportional for width and height, changing both at the same time. The distance between the two touches determines the change in proportion. The values sent for mapping are normalized to the minimum and maximum values for width and height of the Junctions. The default minimum is a 1x1 pixel square and the default maximum is the width and height of the device's screen. Since scaling is proportional, devices with screens of different width and height will constrain the maximum overall size to the smaller edge of the device. Figure 4.11 shows a scaling interaction along with the default maximum values.

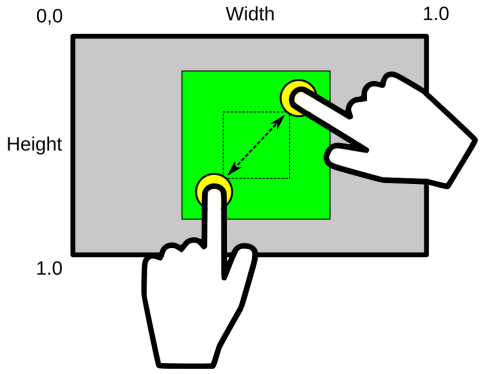
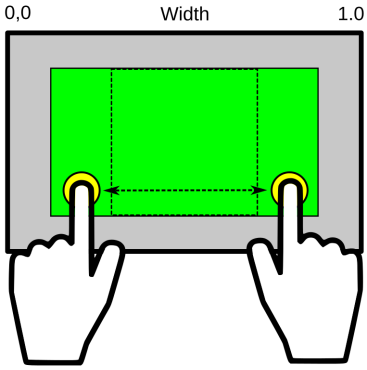


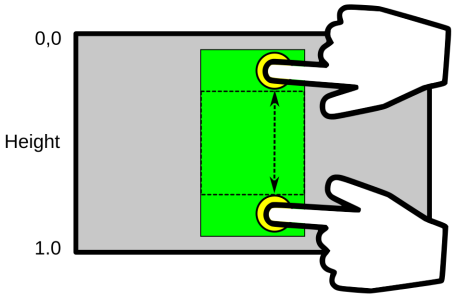
Figure 4.11: Using two touches to scale the width and height of a Junction.

Minimum and maximum width and height can be set to any value and they can be set independently. The sent values for the interactions are then normalized to the new minimum and maximum. Scaling interactions can send both width and height values or just one of the two. Width or height scaling can be disabled, allowing scaling in only one dimension. Figure 4.12a one shows a scaling interaction for width only. In the case of scaling just width or height, only that value is sent. Figure 4.12b shows a scaling interaction for height only.

Figure 4.12: Scaling the width and height of Junctions.



(a) Using two touches to scale the width a Junction.



(b) Using two touches to scale the height of a Junction.

4.3.6 Touches

Touches that occur within Junctions can be mapped to messages. The X and Y location of any touch that occurs within a Junction can be mapped. Each touch message also includes the ID of the touch to distinguish among multiple touches. For mapping the X,Y location of touches, each Junction is assigned its own coordinate space. The top left corner of a Junction represents 0,0 as shown in Figure 4.13. The width and height of the Junction are both assigned a value of one, making the values of the X,Y location that are mapped normalized to the size of the Junction.

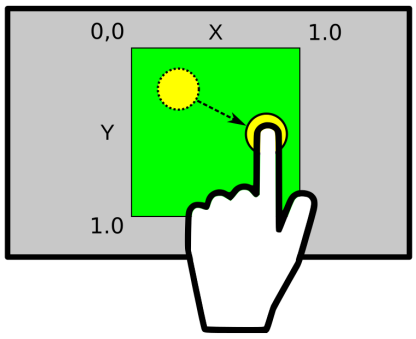
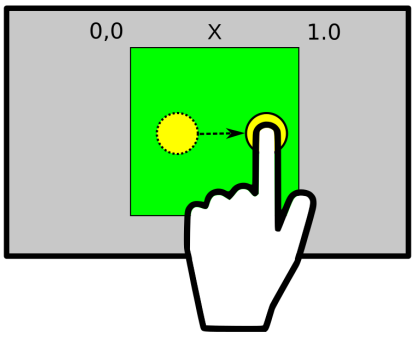


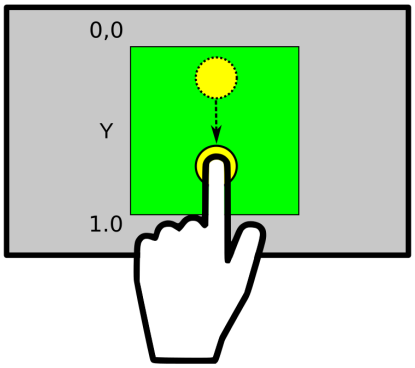
Figure 4.13: One or more touches can have their X,Y location mapped.

This mapping of X,Y locations works well for rectangular Junctions but works less well for elliptical Junction where coordinates make more sense in terms or polar coordinates:

Figure 4.14: Mapping touch X and touch Y inside of Junctions.



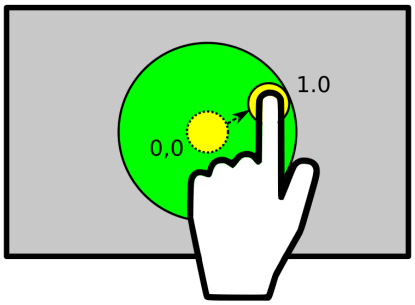
(a) One or more touches can have their X,Y location mapped.



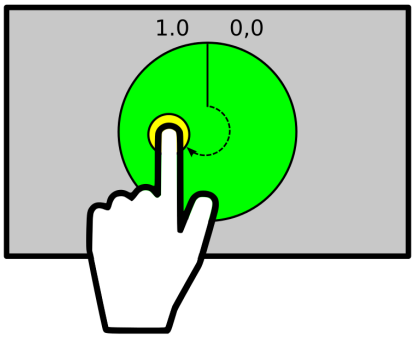
(b) One or more touches can have their X,Y location mapped.

the distance from the center, R, and the angle relative to the center, theta. To solve this problem, Junctions map touch data within ellipses using polar coordinates. The distance from the center is normalized from 0 in the center to 1.0 at the edge of the circle. Figure 4.15a shows a touch at some distance from the center and the 0 and 1.0 normalized points for a circle. The distance calculation works for ellipses with different axis sizes as well as for circles.

Figure 4.15: Mapping touch R and touch theta inside of Junctions.



(a) Touches within ellipses are mapped based on their distance from the center. The touch shown here is moving from the center toward the outer edge.



(b) Contacts within ellipses are also mapped based on their angle. The touch shown here is moving in a clockwise direction.

Touches within elliptical Junctions also send angle data. The top of the ellipse is the starting point for the angle and is normalized from 0 to 1.0 going clockwise around the

ellipse. Figure 4.15b shows a touch and its current angle from the center as well as the 0 and 1.0 angles. The figure shows a circle but the angle calculations work equally well for ellipses.

In addition to the location or angle of touches within Junctions, the number of touches in a Junction can be mapped to messages. With this mapping, a new message will be sent every time the touch count changes. In Figure 4.16, the third touch would send a message with a value of three.

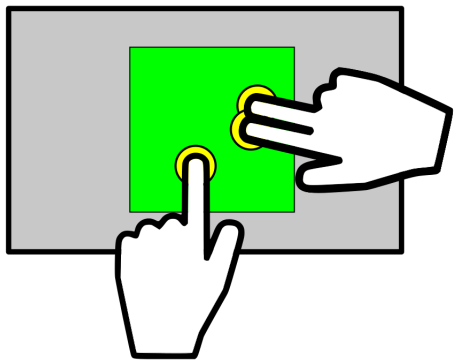


Figure 4.16: The number of touches can be mapped. The mapping shown here would send a value of 3.

4.4 Special Features

In addition to mapping, JunctionBox also has some special features that are related to interactions. The following subsections describe the these additional special features.

4.4.1 Saving Interaction State

JunctionBox has the ability to save the interaction state of Junctions into a file that can then be loaded, returning Junctions to the state when they were saved. Since JunctionBox applications are created with code, it is not essential to store all of the Junction data since much of it is there in the code itself. Instead, the Junction data that is stored is data that is changed via interaction with the interface, data that would not appear in

the code itself. Tools that are similar to JunctionBox allow for the interface itself to be saved but not the state of the interface when interactions have changed it.

The data is stored in an XML-formatted [8] file with the following interaction data for each Junction:

- The Junction's order relative to other Junctions.
- A label for the Junction if it has one.
- The X value for the center.
- The Y value for the center.
- The width of the Junction.
- The height of the Junction.
- The angle of the Junction.
- The toggle state of the Junction.

4.4.2 Inheriting Interactions

Junctions can inherit interactions from other Junctions. When Junctions are added to a parent Junctions, they inherit values for rotation, scaling and translation. This enables child Junctions or sub-Junctions to easily take on any interaction of the parent without direct interaction with the sub-Junctions. It is also possible to allow a sub-Junction to have its own interactions while still inheriting interactions from the parent. Figure 4.17

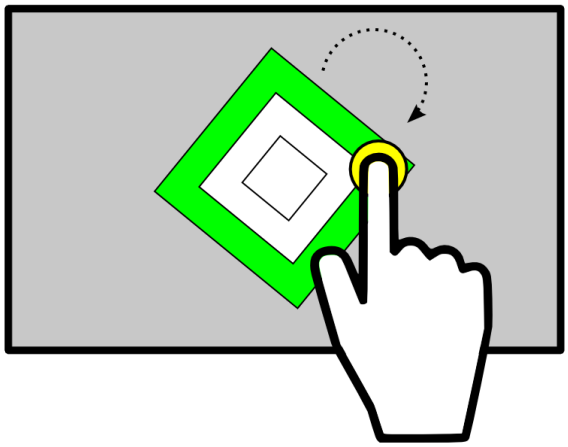


Figure 4.17: Junctions can inherit interactions from other Junctions. The smaller squares in this example are inheriting their rotation angle from the larger parent Junctions shown in green.

shows three square Junctions with the two smaller Junctions acting as sub-Junctions, inheriting their angle from the outer parent Junction.

Beyond, inheriting interactions, sub-Junctions enable the creation of an "interaction graph". This is similar to a scene graph, in which graphical elements in a scene are arranged in a tree so that when a change to the graph must occur, the search for scene elements to change will proceed relatively quickly by not checking every element in the scene. Figure 4.18 shows how an interaction graph works in JunctionBox.

4.4.3 Recording and Playing Interactions

JunctionBox can record and play back interactions with Junctions. The inspiration for this feature comes from options available for audio files. Specifically, audio files represent stored data in a file that can be played back at any time, that can be looped in continuous playback, and can be scaled in time. These same concepts can be applied to multi-touch interactions. In JunctionBox, interactions can be recorded, stored, played back, looped, and time scaled.

For interaction recording, JunctionBox records incoming touch data directly. That

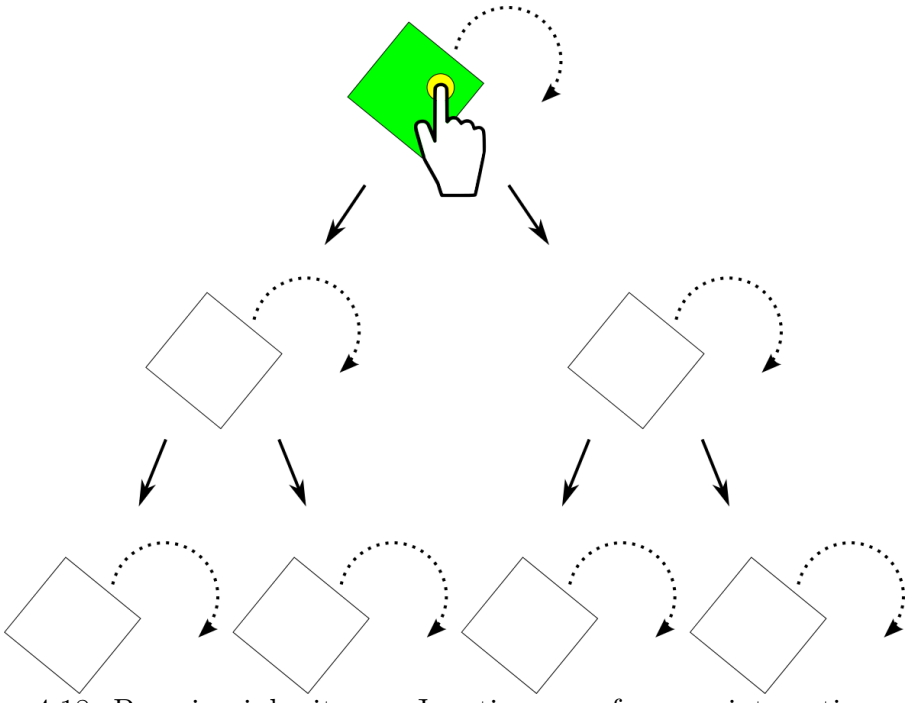


Figure 4.18: By using inheritance, Junctions can form an interaction graph.

data can then be played back, replicating the interaction itself. In order to get an accurate recording of the interaction, timing data is essential. To get accurate times for interaction events, JunctionBox counts the number of nanoseconds between touch events. When recording begins, the time for the first event is zero and then the elapsed number of nanoseconds is recorded for each subsequent event.

The time between the start of recording and the first touch is not recorded. Likewise, any recording done after the last touch is lifted is not included in the recorded interaction. The reason for this is to keep the interaction discreet without having to worry the timing of beginning and ending of recording.

Once recording has been stopped, the interaction data can be played back with the same touch data and timing. Playback is concurrent with any other interactions since playback occurs on a new thread, or subprogram that operates outside of the main interface program. Playback can be stopped at any time.

Any recorded interaction that can be played back can also be looped, that is, played

back repeatedly with looping controlled separately from playback. This allows looping to be started and stopped anytime during playback. While an interaction is playing, starting the looping means that the loop will begin at the end of the current playback. Stopping a loop in the middle of playback will cause the interaction to play to the end but with no subsequent loops.

Once interactions have been recorded, the events in the recording can be scaled in time. By scaling, interactions can be made either faster or slower in time. This feature is analogous to the ability to scale time in sound files. This kind of scaling can allow developers to change the timing on interactions in an arbitrary and creative way.

4.4.4 Connection and Message Management

A major interest that came out of my thesis research is in ways to make computer music performance setup easier. The issue of long setup times for performances is especially acute for distributed music performance systems in which different computers need to communicate to form a coherent system. Each computer then becomes a node in single musical instrument as long as they can easily connect to each other. The management of connections between nodes can be non-trivial and the setup of connections can take time away from rehearsing and performing actual music.

To facilitate the management of nodes in music performance systems, I created the Nexus Data Exchange Format (NDEF) [36, 37], a namespace specification for Open Sound Control (OSC) messages. Appendix B contains the full NDEF specification. NDEF works by providing nodes a *lingua franca* for connecting and for sharing OSC messages. The NDEF specification can be implemented by any system that can handle basic OSC messages. By implementing NDEF, OSC-based systems have a way to identify and connect with other NDEF-supported nodes on either local or remote networks. Once connected via NDEF message exchange, nodes can then request an exchange of the

OSC messages that they accept. The basic functions of the NDEF specification are implemented in JunctionBox.

NDEF is similar to the OSC query system proposal by Schmeder and Wright [103]. The proposed OSC query system offered the ability to explore namespaces by allowing an OSC client to query an OSC server's namespace by including a '/' character at the end of a message pattern. The OSC server would then send back a reply containing any sub-patterns of the provided pattern. The reply begins with the '#' character to distinguish the query system from standard OSC messages. The following is an example from the proposal in which the sub-patterns of /foo/bar are requested. The #reply contains the pattern and the sub-patterns.

```
→ /foo/bar/
← #reply (sss) '/foo/bar/', 'test1', 'test2'
```

An important difference between NDEF and the proposed OSC query system is that NDEF does not require any changes to basic OSC implementations. The '#' character in the query proposal is not allowed in the basic OSC specification [124]. Therefore, the query system cannot be used by implementations of the basic OSC specification. In contrast, NDEF is simply a defined namespace using basic OSC messages that can be implemented by any system without having to change basic OSC functionality.

Libmapper [70] is a software library that allows for a variety of mappings to be made via OSC. But more than just mapping, libmapper also handles connections between nodes on a network as well as message translation. The libmapper library is meant to retain the flexibility of OSC by avoiding the use of a standardized namespace that might allow nodes to automatically communicate. Figure 4.19 shows a GUI interface to the libmapper library with various connections between nodes via OSC message translation.

The problem with this approach is that it simply moves the complexity from OSC

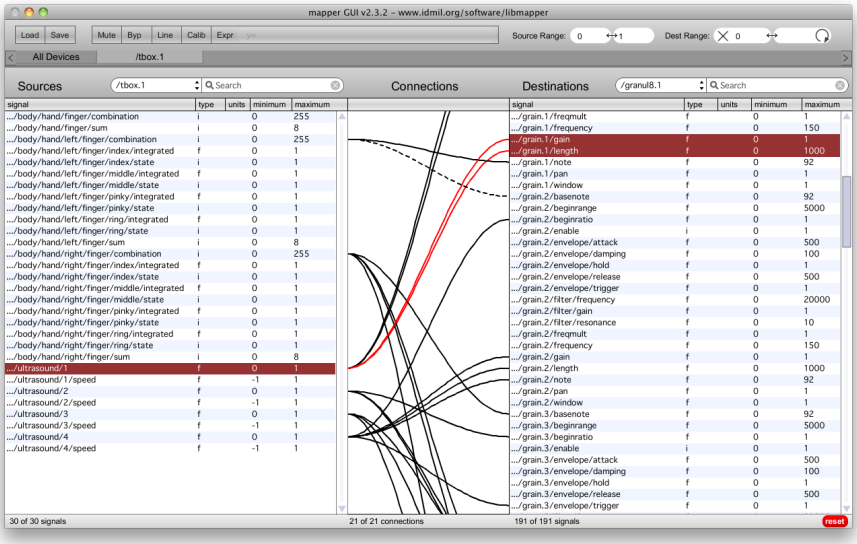


Figure 4.19: A GUI interface for libmapper [70].

and handles it in software. While this sounds appealing, increasing software complexity, especially when various kinds of hardware may be involved means that either the software will not work in enough cases to be truly useful or that configuration of that software will itself be complex. Making configuration more complex is not the goal of NDEF. Rather, NDEF is a way to balance the simplicity of a standardized namespace with enough implementation in software to allow for a variety of connection and message management scenarios.

4.5 Summary

This chapter has described the basics of how the *unit interaction* model is defined and used in the creation of the Junction Box toolkit. While many of the details explained in this chapter are of necessity low level and detailed, together they show the possibility of combining and recombining unit interactions at will. Building from the basis of unit interactions enables simple musical interactions that be combined to create personally chosen complex interactions. The interactions provided by JunctionBox are compared to

similar multi-touch tools in Chapter5 and demonstrated through descriptions of interfaces built with JunctionBox in Chapter 6.

Chapter 5

Comparative Analysis

Science has one methodology, art and design have another. Are we surprised that art and design are remarkable for their creativity and innovation? While we pride our rigorous stance, we also bemoan the lack of design and innovation. Could there be a correlation between methodology and results?

–**Saul Greenberg and Bill Buxton** [40]

In this chapter, I present the results of a comparative analysis of the mappable interactions available in JunctionBox with those available in Control, TouchOSC, and Lemur. The purpose of this comparison is to assess the success of the *unit interaction* model and to place JunctionBox into the context of current multi-touch interaction mapping tools. The criteria that I am using for this analysis is the total number of mappable interactions in JunctionBox versus the other tools. Control, TouchOSC, and Lemur were chosen because they are mapping tools in the same domain as my research and because they offer similar mappable multi-touch interactions.

The comparisons were based strictly on the number of mappable interactions and not on the full feature set in JunctionBox or the other mapping tools since their non-interaction features do not overlap. In other words, JunctionBox has additional features not available in the other tools and those tools have some features not available in JunctionBox. For example, all of the other tools feature MIDI output as an option in addition to Open Sound Control (OSC), which JunctionBox uses exclusively. Though I chose OSC over MIDI for its greater configurability, the reasons for this choice are less important in this context than the fact that MIDI is a communication protocol and not a mappable

interaction. As another example, Lemur includes some physics based feedback, which is not strictly a mappable interaction. At the end of this chapter, in the chapter summary, I mention the additional features that are only available in JunctionBox. Thus this comparison is strictly about overlapping features while focusing on the application of my *unit interaction* model via JunctionBox.

If my *unit interaction* model is successful in finding the fundamental unit interactions inherent in multi-touch, then my model should have more mappable multi-touch interactions. Having more available mappable interactions means that my model is an interaction superset of the interactions available in the other tools. Since JunctionBox is the reification of the model, I am using its multi-touch affordances as the basis for comparison. The chapter is divided into three sections, one for each comparable tool. The first section (5.1) compares Control, the second section (5.2) compares TouchOSC, and the third section compares (5.3) Lemur. The final section (5.4) summarizes the results.

The comparisons presented in this chapter are two-way comparisons and each section has two subsections: 1) a comparison of mappable interactions offered by the tool to JunctionBox and 2) a comparison of JunctionBox mappable interactions to those offered by the tool. Referencing the documentation for each tool, I have used the terms that the documentation uses for the widgets it describes. As part of the analysis I used interfaces built with each tool and made note of the mappable interactions that were available in each widget that the tool offers. Images of each widget, from interface screenshots, are included to show the appearance of the widget and to give an idea of how it is used. The main text for each widget describes the interactions that I found along with equivalent interactions in JunctionBox. At the end of the comparison, I provide a list of the JunctionBox equivalent interactions along with a reference to the figure for the relevant interactions in Chapter 4, Section 4.3.

The second part of each comparison looks at how the interactions offered by JunctionBox compare to the other tool. For this, I created a table listing all of the mappable interactions in JunctionBox along with those in the compared tool. JunctionBox comparison tables contain every interaction from Chapter 4, Section 4.3. A star in the tool's column indicates that a mappable interaction in JunctonBox exists in the tool.

5.1 Control

Control offers a set of widgets that can be customized in terms of their screen placement, size, color, and the message they use for mapping. Each widget has a fixed set of interactions with some having multiple interactions depending on their configuration.

5.1.1 Control to JunctionBox

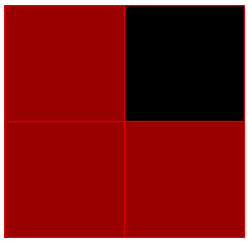


Figure 5.1: A set of four buttons in toggle mode with three of the buttons toggled.

Button

The Button widget has two kinds of interaction: toggle mode and latch mode. There are some additional mapping options with this widget but it has only two basic interactions. In toggle mode, an interaction with the button puts it into a toggled state and a subsequent touch de-toggles the button. Both toggling and de-toggling send messages with the state of the button. A latch-mode button will send a message when it is activated by a touch and then a follow-up message when the touch is removed. The modes of the Button widget are duplicated by the activation and toggling interactions in JunctionBox.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Toggling (Figure 4.5)

MultiButton

The MultiButton widget is a group of Buttons as shown in Figure 5.1. The buttons have the same interactions described for the Button widget.

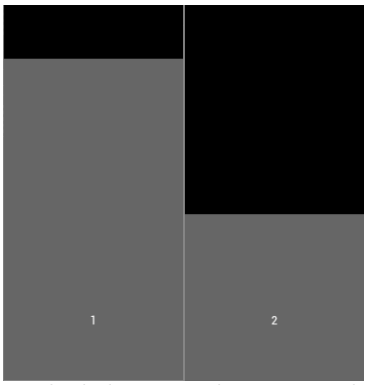


Figure 5.2: Two crossfade Control sliders in the vertical orientation. Slider 1 is moved up relative to slider 2.

Figure 5.3: A normal Control slider in the horizontal orientation with the movable rectangle in the center.

Slider

The Slider widget has two different forms: 1) a crossfader form and 2) a normal form. In the crossfader form, the slider is a small rectangle that is moved to change values. The normal form of a slider is a rectangle that changes size to change values with an increase in size increasing the value and a decrease in size doing the opposite. In either case, sliders can be oriented either horizontally or vertically.

For the normal form, a single touch increases the size of the rectangle while the base of the rectangle stays fixed. Using JunctionBox, this kind of interaction can be replicated in one of two ways. A rectangle can have changes in its width and height mapped. This is not exactly the same interaction used in the Control slider since two touches change the width or height with JunctionBox. To get the same interaction in JunctionBox, the X or Y location of a touch within a Junction can be used to change the width or height with a single touch.

JunctionBox equivalent:

- Scale width (Figure 4.12a) or height (Figure 4.12b)
- Touch X (Figure 4.7a) or Y (Figure 4.7b)

MultiSlider

A MultiSlider widget is a collection of Sliders with same interactions as a basic Slider. Figure 5.3 shows a two-slider MultiSlider.

Figure 5.4: A Control knob.

Knob

The Knob widget is circular and can be rotated in one of two ways: 1) by directly rotating with a touch within its area and 2) by moving a touch up and down in a vertical direction. In either case, the angle of the rotation of the Knob changes its value. A Knob can be moved to a specific value or a value can be selected and the Knob will go directly to that value. The first kind of interactions is replicated in

JunctionBox with the rotation interaction with limits set to duplication the arc-style mapping of the Control Knob. The angle of Junctions can be changed directly, allowing for it to be rotated to a specific value. When an angle is changed to a specific value, messages are still sent in the same way. This kind of direct change to the angle of a Junction can also be used to replicate the second Knob interaction.

JunctionBox equivalent:

- Rotation 1 (Figure 4.10a)

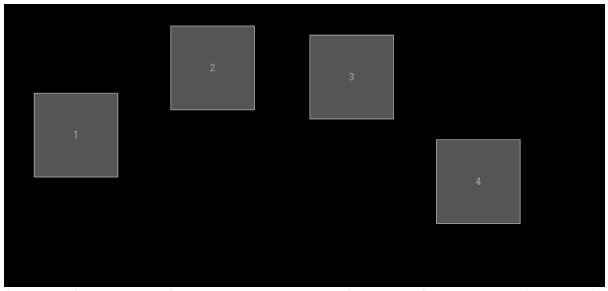


Figure 5.5: A Control MultiTouchXY rectangle. The numbered squares represent the location of touches in the widget.

MultiTouchXY

The MultiTouchXY widgets is a rectangular space that maps the X,Y locations of touches within its area to messages. In JunctionBox, this is done with the touch X and Y mappable interaction.

- Touch X and Y (Figure 4.13)

5.1.2 JunctionBox to Control

Table 5.1 compares mappable interactions in JunctionBox to Control.

Table 5.1: JunctionBox vs. Control

Interaction	**JunctionBox**	**Control**
Activation	★	★
Toggling	★	★
Translation X	★	★
Translation Y	★	★
Translation X and Y	★	
Translation (>1 touches)	★	
Rotation 1	★	★
Rotation 2	★	
Rotation 1 and 2	★	
Rotation Count	★	
Scaling Width	★	★
Scaling Height	★	★
Scaling Width and Height	★	
Touch X	★	
Touch Y	★	
Touch X and Y	★	★
Touch R	★	
Touch Theta	★	
Touch R and Theta	★	
Touch Count	★	

5.2 TouchOSC

TouchOSC offers a variety of widgets that can be mapped to OSC messages. Each widget can have a custom OSC message that it sends when it is activated. The widgets all have a fixed appearance although their sizes can be changed. Widgets are customized with a separate editing application and are then loaded onto a device.

5.2.1 TouchOSC to JunctionBox

The following list describes each TouchOSC [48] widget and the mappable interactions that it affords. Only TouchOSC's interactive widgets are compared since the purpose of the comparison is to compare mappable interactions. TouchOSC has an option to send what it calls Z values for every widget. The letter Z is used because a multi-touch

device has a surface that is a two-axis X-Y plane and the Z axis is the third axis that extends out from the device towards the person performing an interaction. When an interaction begins, a Z message is sent indicating that the widget is active and when an interaction ends, another message is sent indicating the the widget is no longer active. In JunctionBox, this can be done with the activation interaction and so this interaction is listed as a JunctionBox equivalent for every TouchOSC widget.

Push Button

Figure 5.6: TouchOSC push buttons with unpushed on the left and pushed on the right.

The push button widget sends an OSC message when a touch occurs in the area of the button and sends another message when the touch is removed. Button light up when touched and dim again when the touch is removed. The basic functionality of this widget can be done in JunctionBox with the activation mappable interaction. In addition to having a basic activation interactions, this widget also sends a Z message which does exactly what the basic widget does. They both have the same equivalent interaction in JunctionBox.

JunctionBox equivalent:

- Activation (Figure 4.4) for the basic interaction
- Activation (Figure 4.4) for the Z message

Multi-Push

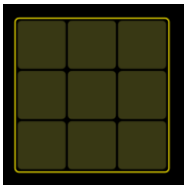


Figure 5.7: TouchOSC multi-push.

The Multi-Push widget is a grid of Push widgets that uses the same mappable interactions.

Toggle Button

Figure 5.8: TouchOSC toggle buttons with toggled on the left and de-toggled on the right.

The toggle widget has two touch steps: 1) a touch to set the widget to a toggled state and 2) a touch to de-toggle the widget. The widget lights up when it is in the toggle state and dims when it is de-toggled. Messages are sent for when the state is changed with one message for toggle and one for de-toggle. This interactions is equivalent to the toggling interaction in JunctionBox.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Toggling (Figure 4.5)

Multi-Toggle

The multi-toggle widget is a collection of toggle widgets laid out in a grid. The mappable interactions are the same as they are for toggles except that each toggle is assigned a number in the grid.

Figure 5.9: TouchOSC multi-toggle.

Fader/Rotary

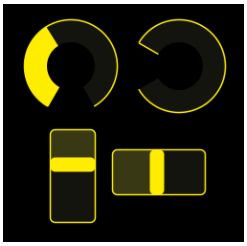


Figure 5.10: TouchOSC faders and rotaries.

Faders feature a small rectangle that is moved within the confines of a larger rectangle. A fader can be oriented vertically or horizontally with the interaction mapping in the Y direction or the X direction depending on orientation. In JunctionBox, these interactions are enabled with the translate X or translate Y interactions.

Rotaries are similar to faders but instead of moving a rectangle, the interaction depends on the angle of a touch relative to the center of a circle. When a touch happens within the widget, a circular strip changes grows or shrinks within the circle to indicate an increase or decrease in value. Clockwise movement increases the value and counter-clockwise movement decreases the value. The rotation interaction in JunctionBox is the same kind of mappable interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)

- Translate X (Figure 4.7a) or Y (Figure 4.7b) for the Fader
- Rotation 1 (Figure 4.10a) for the Rotary

Multi-Fader

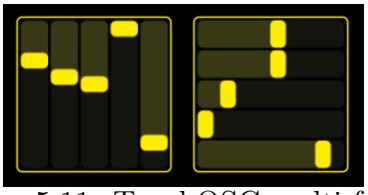


Figure 5.11: TouchOSC multi-fader.

A Multi-Fader combines a number of Fader widgets into a single widget.

Encoder

Figure 5.12: TouchOSC encoder.

An encoder is a rotatable widget with no limit on the angle or the number of rotations. The rotation interaction in JunctionBox offers the same mappable interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Rotation (Figure 4.10a)

XY Pad

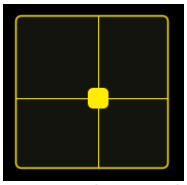


Figure 5.13: TouchOSC XY pad with target lines that show the location of the last touch.

An XY pad takes the X,Y location of any touch in its rectangular area and includes those values in the message set for the widget. This is done in JunctionBox with the touch X and Y interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Touch X and Y (Figure 4.13)

Multi-XY

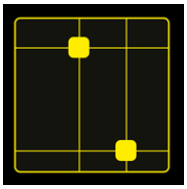


Figure 5.14: TouchOSC multi-XY with two touches.

The XY widget works in the same was as the XY Pad but it will take up to five touches. Otherwise, the mappable interaction is the same as for the XY Pad widget.

5.2.2 JunctionBox to TouchOSC

Table 5.2 compares the mappable interactions available in JunctionBox to those available in TouchOSC.

Table 5.2: JunctionBox vs. TouchOSC

Interaction	**JunctionBox**	**TouchOSC**
Activation	★	★
Toggling	★	★
Translation X	★	★
Translation Y	★	★
Translation X and Y	★	
Translation (>1 touches)	★	
Rotation 1	★	★
Rotation 2	★	
Rotation 1 and 2	★	
Rotation Count	★	
Scaling Width	★	
Scaling Height	★	
Scaling Width and Height	★	
Touch X	★	
Touch Y	★	
Touch X and Y	★	★
Touch R	★	
Touch Theta	★	
Touch R and Theta	★	
Touch Count	★	

5.3 Lemur

Lemur is similar to both Control and TouchOSC in terms of offering mappable widgets. Unlike the other widget tools, Lemur allows "skins" that can customize the appearance of widgets. A notable feature of Lemur is the use of physics to animate the movement of certain objects after they are released. When objects are released, they can bounce, rebound, and oscillate on their own. Each of these physical movements is mapped to OSC messages. For this analysis, I have not included physics as a mappable interaction in the comparison. The reason for excluding physics is that this is not strictly speaking an interaction since the objects with physics move on their own, after the actual interaction has occurred. The physics in Lemur represent a mappable animation. The strict focus of this analysis is to compare mappable interactions rather than mappable animations.

5.3.1 Lemur to JunctionBox

The following widgets (objects) are available in Lemur [67]. Details about the widgets are taken from the Lemur User Guide, Chapter 12 [66] and from building an interface using those widgets to test the interactions and mapping. Like the Z messages in TouchOSC, nearly all of the widgets in Lemur send a message indicated that they are actively being used and another message when they are no longer active. This is the equivalent of the activation interaction in JunctionBox. The list of equivalent interactions for these widgets references this JunctionBox interaction.

Pad

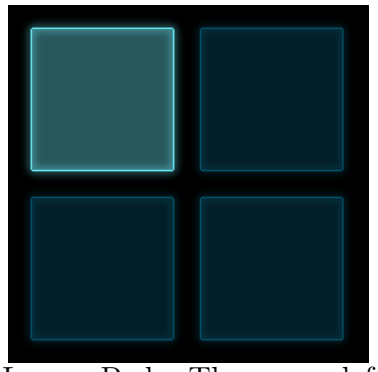


Figure 5.15: Four Lemur Pads. The upper left Pad is activated.

Pad widgets are triggered by a touch like Control's Button in "latch" mode and TouchOSC's Push Button. A touch sends a message when it contacts the widget and another when it leaves the widget.

JunctionBox equivalent:

- Activation (Figure 4.4)

Switch

Figure 5.16: Two Lemur Switches. To left Switch is not toggled and the right Switch is toggled.

The Switch widget is like the Control Button in "toggle" mode and the TouchOSC Toggle Button. When a touch occurs in its area, it is set into a toggled state. Touching again de-toggles the widget.

JunctionBox equivalent:

- Toggling (Figure 4.5)

StepNote

A StepNote is a set of toggle switches that offers the same interaction described for the Switch widget.

JunctionBox equivalent:

- Toggling (Figure 4.5)

Custom Button

The Custom Button can be either a Pad or a Switch. Neither option changes the mappable interactions.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Toggling (Figure 4.5)

Fader

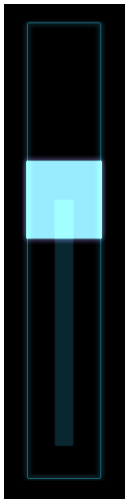


Figure 5.17: Lemur Fader in a vertical orientation. The bright blue rectangle moves to change values.

Like the Slider in Control and the Fader in TouchOSC, Lemur's Fader widget is a movable rectangle that changes the value based on its movement. Fader's can be oriented vertically or horizontally.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Translation X (Figure 4.7a) or Y (Figure 4.7b)

Knob

The Knob widget has two rotation modes: 1) a constrained rotation mode and 2) an encoder mode with no constraints. This very similar to the knob widget in Tou-

Figure 5.18: Lemur Knobs in constrained mode on the left and in encoder mode on the right.

chOSC. Like that widget, JunctionBox does the same mapping with the rotation interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Rotation 1 (Figure 4.10a) for both modes

MultiBall

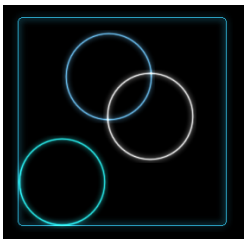


Figure 5.19: Lemur MultiBall with thee balls.

The MultiBall widget is one or more movable circles inside of a rectangle. When the locations of the circles are changed, they send messages with their current location. This interaction can be replicated in JunctionBox with the touch X and Y interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Touch X and Y (Figure 4.13)

MultiSlider

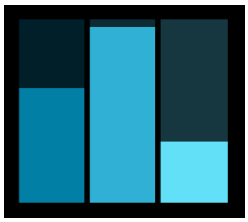


Figure 5.20: Lemur MultiSlider in a vertical orientation with the sliders set to various values.

The Lemur MultiSlider is a set of slider objects. A slider in a Lemur interface is distinct from a Fader in that it does not use a movable rectangle to change values. Rather, a rectangle changes size to set values. Either the height or the width of a slider rectangle changes, depending on whether it is in a horizontal (width) or vertical (height) orientation.

JunctionBox equivalent interactions:

- Activation (Figure 4.4)
- Scaling Width (Figure 4.12a) or Height (Figure 4.12b)

StepSlider

The StepSlider is similar to the MultiSlider but it sends all of the values whenever there is an interaction. Otherwise, the interactions are the same.

JunctionBox equivalent:

- Activation (Figure 4.4)

- Scaling Width (Figure 4.12a) or Height (Figure 4.12b)

Range

Figure 5.21: Two Range widgets where the size of the rectangle represents the size of the value range. The widget on the right has a larger range.

The Range widget sets a range of values based on the width or height of a rectangle. Increasing the width or height increases the range of values while decreasing it does the opposite.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Scaling Width (Figure 4.12a) or Height (Figure 4.12b)

RingArea

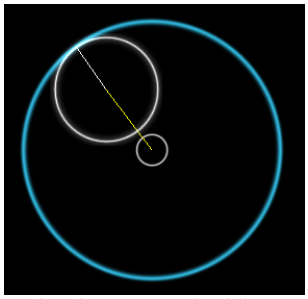


Figure 5.22: Lemur RingArea with the Ring held in the upper left part of the circle.

The RingArea widget allows a ball to be moved around inside a circle. Moving the ball changes its X and Y position inside the circle and this is then mapped to a message. More interesting than the basic interactions is that this widget actually has its own physics in that, when released, it bounces around the center of the circle until it runs out of momentum and stops moving. For the RingArea, messages are still sent after the ball is released as it loses momentum. JunctionBox does not have physics but the basic interaction can be mapped with the touch X and Y interaction.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Touch X and Y (Figure 4.13)

Breakpoint

The Breakpoint is an envelope editor widget. An envelope is applied to a sound to gradually increase its gain with an optional decrease in gain followed by an optional sustained gain and ending with a gradual decrease in gain. Envelopes are the equivalent, when done in this way, of pressing a key on a keyboard, holding

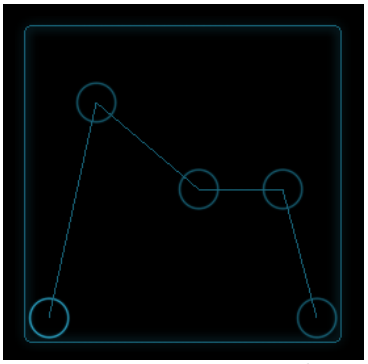


Figure 5.23: Lemur BreakPoint.

the note for that key, and then lifting the key again to end the note. The envelope is represented in the BreakPoint with at least three circles, one for each change in gain. Each the the circles can be moved, changing the shape of the envelope. When the circles are moved, messages containing the X and Y location of the circle are sent. Figure 5.23 shows the BreakPoint widget with five circles for the envelope.

JunctionBox equivalent:

- Activation (Figure 4.4)
- Touch X and Y (Figure 4.13)

5.3.2 JunctionBox to Lemur

Table 5.3 shows the comparison of mappable interactions in JunctionBox to those in Lemur.

5.4 Summary

The point of this comparative analysis was to determine whether JunctionBox offers more mappable interactions compared to similar mapping tools, making JunctionBox an interaction superset. Table 5.4 summarizes the comparisons among Control, TouchOSC, Lemur, and JunctionBox. As the comparisons show, JunctionBox can replicate any of the

Table 5.3: JunctionBox vs. Lemur

Interaction	**JunctionBox**	**Lemur**
Activation	★	★
Toggling	★	★
Translation X	★	★
Translation Y	★	★
Translation X and Y	★	
Translation (>1 touches)	★	
Rotation 1	★	★
Rotation 2	★	
Rotation 1 and 2	★	
Rotation Count	★	
Scaling Width	★	★
Scaling Height	★	★
Scaling Width and Height	★	
Touch X	★	
Touch Y	★	
Touch X and Y	★	★
Touch R	★	★
Touch Theta	★	★
Touch R and Theta	★	★
Touch Count	★	

mappable interactions in Control, TouchOSC, or Lemur. JunctionBox also offers more mappable interactions in total, making it a superset of the interactions in the comparable tools. In other words, JunctionBox could be used to build any of these widget-based tools while at the same time offering more interaction options. By showing that JunctionBox is an interaction superset of the other tools, I have shown the *unit interaction* model that I developed during my research has found more fundamental multi-touch interactions thus meeting my research challenge of finding the universal building blocks inherent in multi-touch.

The additional special features in JunctionBox that are beyond those contained in Control, TouchOSC, and Lemur that are not used for this comparision include: 1) the ability to save and load ineraction states, 2) allowing actions to be inheritable, thus supporting propogation of interactions, 3) incoporating the ability to record, play back,

loop, and time stretch interactions and 4) offering connection and message management functionality that makes it easier for networked computers to connect and to share the OSC messages that they use for mapping.

Table 5.4: JunctionBox vs. Control, TouchOSC, and Lemur

Interaction	**JunctionBox**	**Control**	**TouchOSC**	**Lemur**
Activation	★	★	★	★
Toggling	★	★	★	★
Translation X	★	★	★	★
Translation Y	★	★	★	★
Translation X and Y	★			
Translation (>1 touches)	★			
Rotation 1	★	★	★	★
Rotation 2	★			
Rotation 1 and 2	★			
Rotation Count	★			
Scaling Width	★	★	★	★
Scaling Height	★	★	★	★
Scaling Width and Height	★			
Touch X	★			
Touch Y	★			
Touch X and Y	★	★	★	★
Touch R	★			
Touch Theta	★	★	★	★
Touch R and Theta	★			
Touch Count	★			

Chapter 6

Interfaces and Performances

Musical interface construction proceeds as more art than science, and possibly this is the only way that it can be done.

–**Perry Cook** [20]

Digital lutherie is in many senses, very similar to music creation. It involves a great deal of possible and different knowhow, the use of many technical and technological issues but, like in music, there are no inviolable laws. That is to say that digital lutherie should not be considered as a science, no more than music, but as a sort of craftsmanship, that may sometimes produce - in very exceptional cases - a work of art; no less than music.

–**Sergi Jordà** [55]

In this chapter, I describe a series of interfaces that I built with JunctionBox and used in various performance situations (including one installation). Through these interfaces and their associated performances, I demostrate the success of the *unit interaction* model in real performance situations. Each section in the chapter describes an interface, the mappable interactions used with that interface, and the how the interactions were used to control audio. A screenshot of the interface opens each section. This is followed by a list of who created each interface and when and where each interface was performed. A list of mappable interactions at the end of each description points to the relevant figure in Chapter 4, Section 4.3 that describes that interaction. The chapter ends with a short summary (6.7).

I designed all but one of the interfaces myself and coded all of them using JunctionBox. One interface, Under Control (described in Section 6.5), was a collaborative design with Simon Fay. In addition, I programmed the audio engines for all but the Under Control interface. The instruments (interface and audio engine combined) that I designed myself were intended to allow me to experiment with musical interfaces both visually and sonically. These experiments show just some of the creative possibilities that JunctionBox offers.

My own research falls into the general category of digital lutherie or building musical instruments with computer software and hardware. Digital lutherie certainly has an element of craftsmanship but there is more to the story of digital lutherie than just craftsmanship. In order to craft an instrument, especially a work of art, the builder must have the right tools. This is especially true for software tools like libraries in which the library itself actually becomes a part of the instrument. A well-designed library or toolkit both addresses the need for basic building blocks and at the same time addresses the need for creative freedom, the beginning of any craftsmanship. To determine whether a tool deisgn is successful, the design of a tool can be tested by using it in practice.

6.1 Apollo 20

Created by **Lawrence Fyfe**

Performed at **CCRMA Summer Workshop**

July 30, 2010

Stanford University

In July of 2010, I co-taught, with Adam Tindale, a one-week CCRMA Summer workshop on using JunctionBox to create multi-touch musical interfaces. For the workshop, we used a custom-built FTIR multi-touch table as the touch hardware. During the workshop,

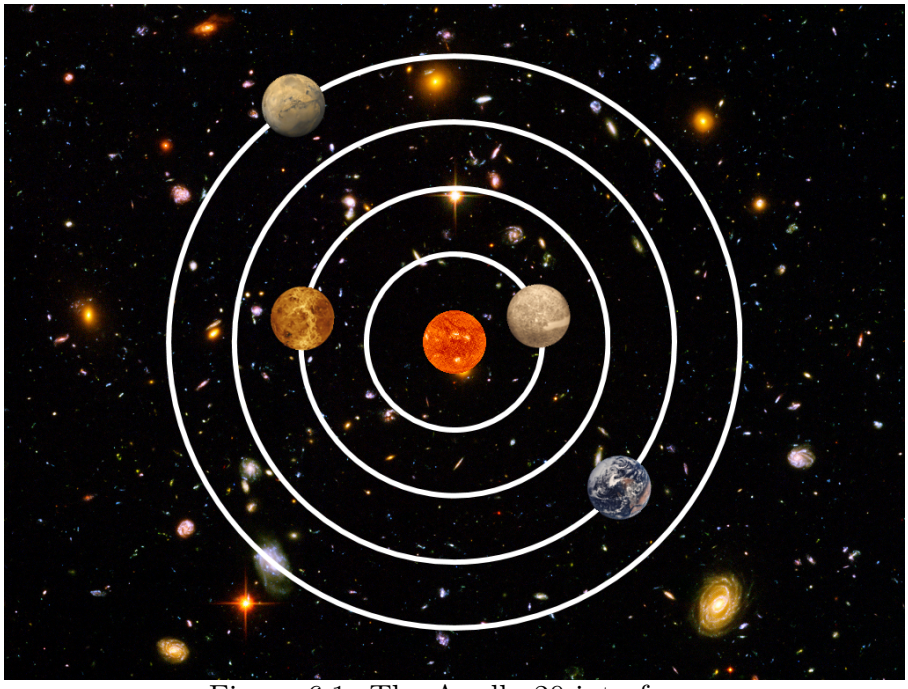


Figure 6.1: The Apollo 20 interface.

I built my own interface called Apollo 20. At the end of the workshop, the instructors and the students put on a short concert to demonstrate their work with JunctionBox. Figure 6.1 shows the Apollo 20 interface. My artistic intention with this interface was to experiment with constraint by creating a very simple set of interactions and to see how they can be used.

The interface features four concentric circles designed to represent planetary orbits much like an orrery [123] represents planets in the solar system. Each orbit has a planet, representing the four inner planets in the solar system: Mercury, Venus, Earth and Mars. The four planets are toggle buttons and each uses an actual image of that planet. The toggle buttons control a bank of four FM synthesizers, one for each planet. The orbital areas are rotatable and the planets move as the orbital area is rotated. Rotating the orbital area de-tunes the corresponding FM synthesizer. The mapping between this interface and the audio engine is described in the next section since these two interfaces share the same audio.

Mappable interactions:

- Toggling (4.5)
- Rotation (4.10a)

6.2 Orrerator

Figure 6.2: The Orrerator interface based on an orrery metaphor.

Created by **Lawrence Fyfe**

Performed at **CEC 25th Anniversary Concert**

November 22, 2011

University of Calgary

Recording played at **Linux Audio Conference**

April 13, 2012

Stanford University

The Orrerator is a revision of the Apollo 20 interface described previously. Like Apollo 20, the metaphor for the Orrerator is the orrery or a model of the solar system, as shown in Figure 6.2. For this interaction, I wanted to experiment with a more abstract representation for the visuals and to add more controls for the audio to give the interface more expressive control after the experiment in constraint in Apollo 20. The Orrerator interface has widget-like controls but with a much more stylized appearance.

I performed live with the Orrerator at the CEC (Canadian Electroacoustic Community) 25th Anniversary Concert. The Orrerator is designed both as a general instrument that could be used for a variety of pieces but also specifically for my composition entitled *Sol Aur*. For *Sol Aur*, I recorded audio of myself playing the instrument and that recording was played at the 2012 Linux Audio Conference.

The Orrerator uses the same audio engine as Apollo 20, programmed in PureData (Pd), and features four FM oscillators. The interface has four planet buttons that light up when toggled by simply touching the particular planet. Each planet button turns a single FM oscillator on or off. In addition to on or off controls, each planet button can be rotated around its orbit by touching the orbit area and rotating it around the center. The planets move when the orbit area is rotated, giving visual feedback. The change of rotation will detune that particular FM oscillator from its base frequency by a factor of between one and two with the starting vertical position being one and a full rotation back to that same position being two. Orbital rotations are limited to 360 degrees from the starting position. That is, planets start at the top of the interface and can be fully rotated back to that position. They can also be rotated counter-clockwise to their starting position. On the left and right edges of the interface are two sliders: the left slider changes the index of modulation and the modulation frequency and the right slider changes the gain of all four oscillators at the same time.

Mappable interactions:

- Toggling (4.5) for the planets
- Rotation (4.10a) for the orbital areas
- Translation Y (4.7b) for the sliders

6.3 Particulator

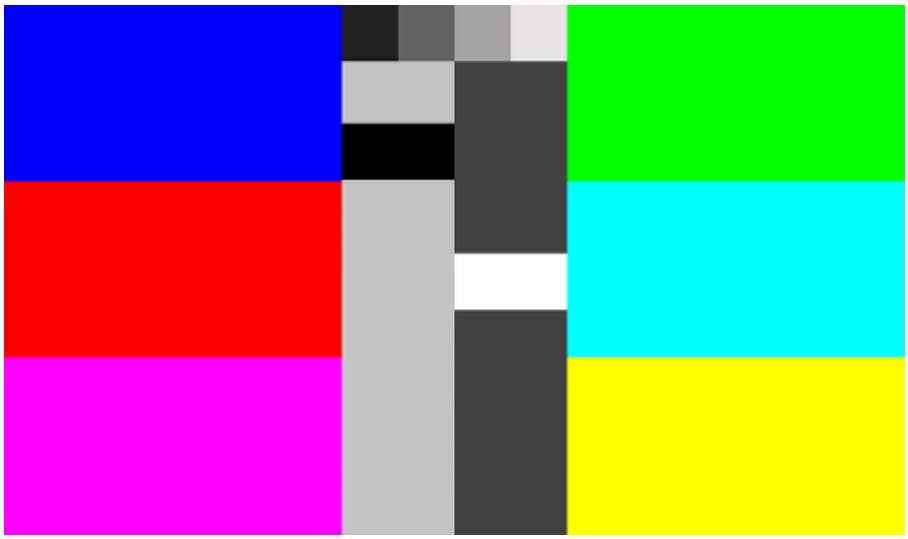


Figure 6.3: The Particulator.

Created by **Lawrence Fyfe**

Performed at **New Music Ensemble Final Concert**

December 3, 2012

University of Calgary

The design of the Particulator interface is inspired by the test pattern that was used for analog television broadcasting. Test patterns are a representation of the color spectrum,

split into discernible parts. This is somewhat analogous to the breakup of an audio file into parts in granular synthesis. The audio engine has two granular synthesizers and is built in Pd. For each granular synthesizer, I used two audio files, each taken from my recording of *Sol Aur* mentioned in the description of the previous interface.

I performed with the instrument at the final concert of the New Music Ensemble course at the University of Calgary. For that concert, I worked with the students to develop a semi-improvised performance featuring each of their computer-based instruments while I performed with the Particulator.

The three large rectangles on the right and left control, from top to bottom, the position, the duration, and the interonset times for the granular synthesizers. The middle sliders control the gain of each audio granulator (they are directly next to their corresponding synth controls). The middle buttons on top of the sliders from left to right, turn on the granulators for continuous playback and fire off a single grain respectively.

Mappable interactions:

- Activation (4.4) for firing single grains
- Toggling (4.5) for controlling continuous playback
- Translation Y (4.7b) for the sliders
- Touch X and Y (4.13) for the granular parameters

6.4 Glass Steps

Created by **Lawrence Fyfe**

Performed at **Interactions Lab Demo Day**

December 9, 2013

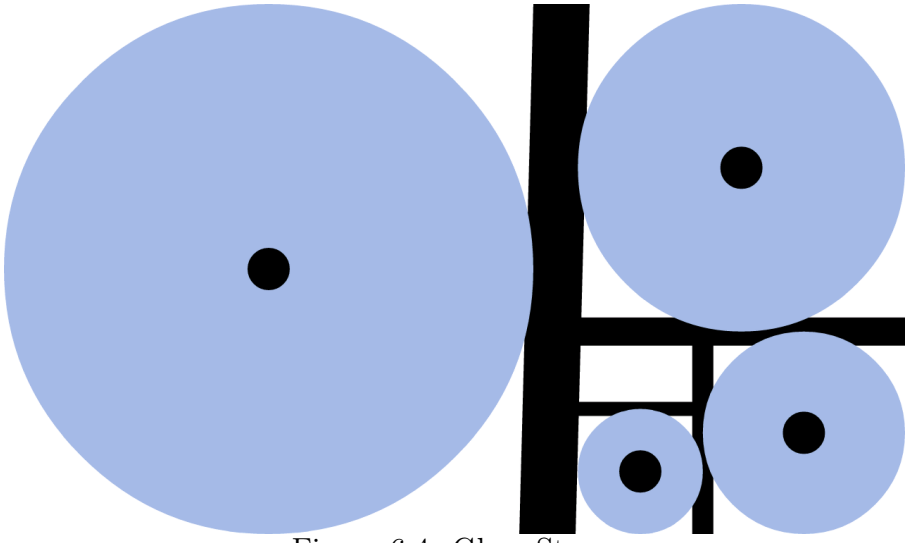


Figure 6.4: Glass Steps.

University of Calgary

Glass Steps originated with my interest using the golden ratio [120] for both visuals and audio. The steps (blue circles) in the interface are sized according to the golden ratio with each set into a rectangle that relates to a larger rectangle as shown in Figure 6.5.

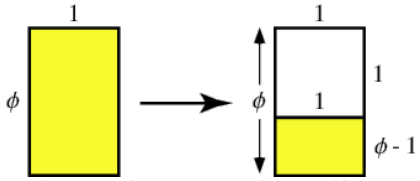


Figure 6.5: The golden ratio relating larger and smaller yellow rectangles. Image taken from Mathworld [120].

Each step plays a particular note and the notes are also tuned using the golden ratio. The largest circle plays the lowest note and each successive note is 1.6 (close to the value of the golden ratio, ϕ) times the frequency of the previous note.

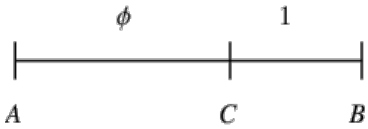


Figure 6.6: ϕ represents the golden ratio. Image taken from Mathworld [120].

As the steps get smaller, the frequency gets higher in an approximation of the notion

of smaller acoustic instruments have higher frequencies than larger ones. I used this interface for short performances during the Interactions Lab demo day where people visit the lab and the students demonstrate their work for the visitors.

The audio engine used for Glass Steps is ChucK [16] which I chose because ChucK makes it easy to implement physical models of acoustic instruments. Each of the circles controls a banded wave guide [17] physical model that sounds like a glass harmonica. The distance of touches from the center of the circle controls the gain of each glass harmonica, with the highest gain closest to the center of the circle and the quietest closest to the edge. A toggle button in the center of each circle controls a chorus effect. The number of touches in each circle controls the number of voices in the chorus effect.

Mappable interactions:

- Activation (4.4)
- Toggling (4.5) for the chorus effect
- Touch R (4.15a) for the gain
- Touch Count (4.16) for the voices

6.5 Under Control

Created by **Lawrence Fyfe and Simon Fay**

Performed at **International Computer Music Conference - Sound and Music Computing (ICMC-SMC)**

September 16, 2014

Athens, Greece

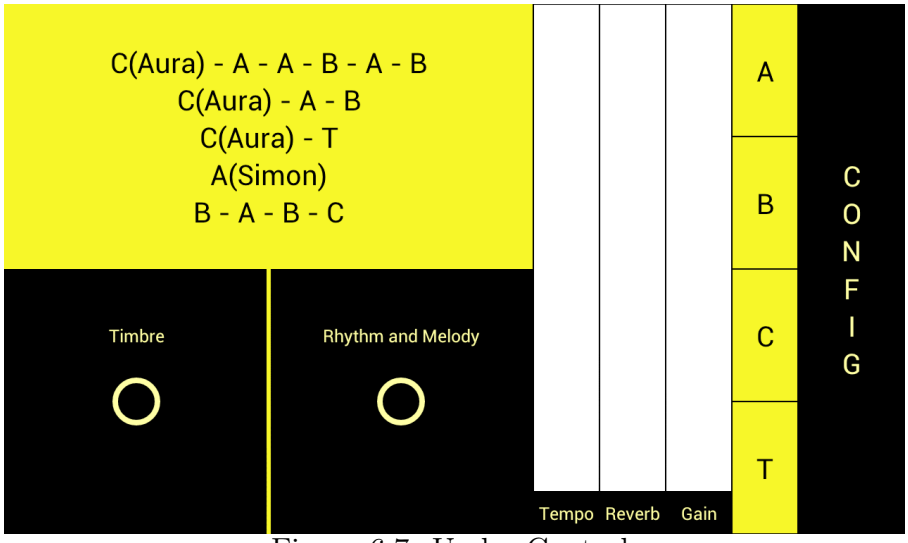


Figure 6.7: Under Control.

Under Control was built in collaboration with Simon Fay for his piece *_under_scored_* performed with the Aspect ensemble [34] (myself, Simon Fay, and Aura Pon) at the joint ICMC-SMC 2014 conference. I formed the ensemble with Simon and Aura to create a performance vehicle for our various forms of research into computer music. For my own research, the Under Control interface is built with JunctionBox. I played this interface while Simon played guitar and Aura Pon played oboe. The interface controlled a Max/MSP patch developed by Simon that featured a continuously playing algorithmic synthesizer with a strong rhythmic component since the piece is in a Jazz fusion style.

The piece is a structured improvisation with four sections: A, B, C, and T (for transition). In the upper left corner is a list of the sections of the piece that allowed me to keep track of the current section. Solos are marked with names, either Aura or Simon. The section is changed during the course of the piece by selected one of the A, B, C, or T buttons on the right side. Whenever I changed the section, a simple visual score indicated to Simon and Aura both the current section and the upcoming section that I had selected. The score is shown in Figure 6.8.

For the different sections, I changed various parameters to give each section a par-

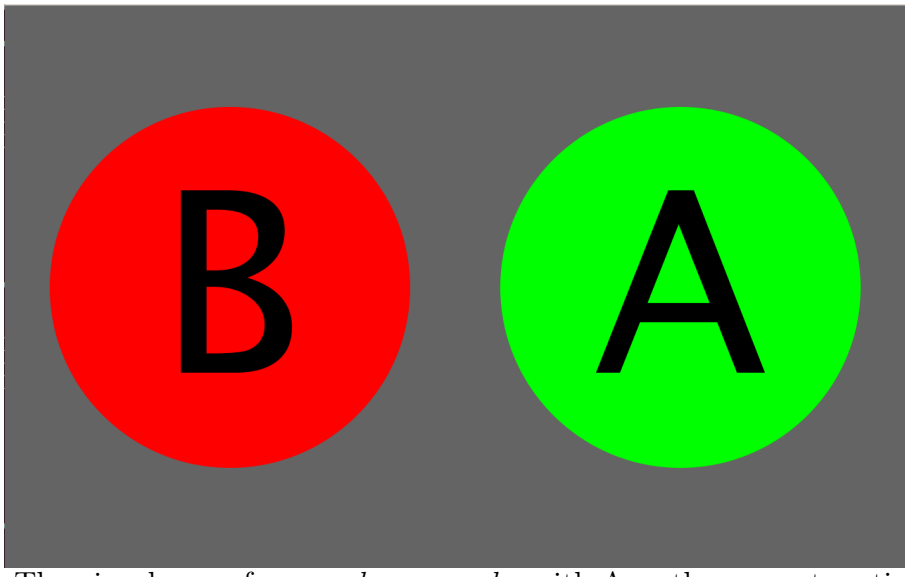


Figure 6.8: The visual score for *_under_scored_* with A as the current section and B as the upcoming section.

ticular feel. The two sections marked "Timbre" and "Rhythm and Melody" track the X and Y location of touches that are mapped to those parameters in the Max/MSP patch. When moving from one corner to another, the timbre, for example, is cross-faded between timbres as the touch moves to the other corner. Therefore, the center, indicated by a ring, represents all four timbre values mixed equally. The mapping is the same for changes to rhythm and melody. A set of sliders controls the labelled parameters. The large "Config" button on the right side of the interface opens a form for changing the IP address and port number used to send messages to the audio engine.

Mappable interactions:

- Activation (4.4)
- Toggling (4.5) for the section buttons
- Translation Y (4.7b) for the sliders
- Touch X and Y (4.13) for the timbre and rhythm and melody controls

6.6 Distance 2

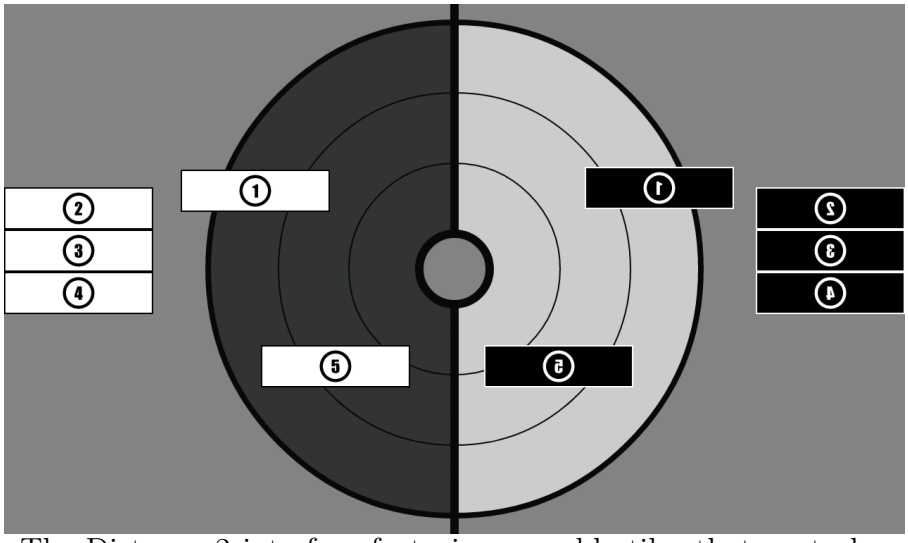


Figure 6.9: The Distance 2 interface featuring movable tiles that control various audio files.

Created by **Lawrence Fyfe**

Installed at **Interpreter: The Abstraction of Narrative through Interactive Digital Rendering Systems**

February 6-27, 2015

GalleryFM, University of Calgary

The Distance 2 interface was used for my installation entitled *Distance 2 (Toshi Ichiyanagi)*. Distance 2 was inspired by Toshi Ichiyanagi's composition *Distance* [50] in which the performer must be a fixed distance away from the instrument during the performance. For my installation, the notion of distance was conceptual rather than actual. The sounds I used for this installation are all taken from an interview with Toshi Ichiyanagi with some effects added. The conceptual distance is from Toshi Ichiyanagi's words as the sounds are changed as described below.

A series of numbered tiles, ten in total, can be moved across the interface with each tile representing a snippet of the interview with Toshi Ichiyanagi. Figure 6.9 shows the

tiles on the interface. When the tiles are placed in the target circle and the touch used to move the tile is released, the sound for that tile begins to play. Touching a tile while it is playing will pause playback. White tiles are the original sound and black tiles represent the sound reversed. The left side of the target plays sound in the left speaker and the right side plays in the right speaker.

The concentric circles in the target represent the playback speed for the chosen sound tiles. The very center plays back the sound at normal speed. Each larger concentric circle slows down the playback speed by a set amount with the outer circles playing at 2x, 4x, and 8x the normal playback speed respectively. Stretching the files in this way significantly alters the sound as the playback rate is slowed, representing a conceptual distance from the original sound. The audio engine for this piece is written in Pd with a custom file playback control mechanism designed by me.

Mappable interactions:

- Activation (4.4) for the playback
- Translation X and Y (4.6) for the placement of the tiles

6.7 Summary

This chapter described a series of musical interfaces that I built using JunctionBox and used in performance and installation situations. By using JunctionBox-built interfaces in a series of performances, I have shown that my *unit interaction* model is usable in real musical situations. The interfaces also serve to provide some idea of the range of creative possibilities that my model/JunctionBox affords.

Chapter 7

Design Principles

> *I also discovered that in the grand scheme of things, there are three levels of design: standard spec, military spec and artist spec. Most significantly, I learned that the third, artist spec, was the hardest (and most important). If you could nail it, everything else was easy.*
>
> –**Bill Buxton** [12]

In this chapter, I describe the design principles that I distilled from my research in applying my *unit interaction* model via the JunctionBox toolkit. The principles presented here meet my research challenge of distilling my research into usable design principles. The value of these principles lies in their ability to meet "artist spec", as Buxton calls it, in designing toolkits for building multi-touch musical interfaces. In the context of my research, I define "artist spec" as doing two important things: 1) offering creative coding functionality that addresses multi-touch in a musical context and 2) allowing creative coders the freedom to build multi-touch musical interfaces according to their own creative whims.

I distilled the design principles in both building and using JunctionBox to build musical interfaces for my own creative practice. Each of the principles is a variation on the concept of tolerance. The chapter is divided into five sections with the first four describing a specific kind of tolerance that is relevant to the design of JunctionBox. The first section describes interaction tolerance (7.1), the second describes mapping tolerance (7.2), the third describes networking tolerance (7.3, and the fourth describes graphical tolerance (7.4). The last section (7.5) is a short summary of the chapter.

The overarching principle that I distilled from my research is *tolerance*. My motivation for using this term for my design principles comes from the definition of the verb form, *tolerate*:

> **tolerate**: [**tol**-uh-reyt] to allow the existence, presence, practice, or act of without prohibition or hindrance. [31]

In the context of my research, tolerance means allowing programmers the freedom to build highly customized interfaces while providing relevant functionality. A toolkit, JunctionBox included, offers functionality that inevitably leads to some creative constraints. The trick in designing a toolkit is to balance functionality with tolerance in how that functionality is used. My goal in developing a unit interaction model through JunctionBox was to enable a high degree of tolerance in balance with a rich set of multi-touch mappable interactions. The importance of the design principles in this chapter is that they serve to bring that balance to the fore for anyone who wants to build a mapping toolkit that combines interaction, mapping, and graphics.

7.1 Interaction Tolerance

The first principle that I derived from my research in designing and using JunctionBox is interaction tolerance. This means that a toolkit should allow for the widest possible range of interaction options to choose from when building a multi-touch musical interface. A builder should have complete control over what interactions are or are not used for a given interface. This means that a builder can use any interaction or combination of interactions to build interfaces that are as simple or as complex as desired. This principle is very much in line with the universal building block philosophy that Max Mathews used in designing unit generators for audio programming. As such, interaction tolerance is the key to implementing my *unit interaction* model and JunctionBox demonstrates this

principle in the interactions that it offers interface builders. These are listed in Chapter 4, Section 4.3.

7.2 Mapping Tolerance

Mapping tolerance is about allowing but not requiring any multi-touch interaction to be mappable. Once a set of interactions are selected for a given interface, all of those interactions should be mappable to messages if desired. Once mapping is desired for a given interaction, it should be possible to enable the mapping by selecting a message for it. There should be total freedom in selecting the message for any given mapping. Messages are the glue that connects interfaces to audio engines to form a single musical instrument. OSC is an inherently flexible message system, allowing builders to choose their own messages. A toolkit that uses OSC should not get in the way of the flexibility in OSC even while providing functionality that makes OSC easier to use. In JunctionBox, a builder can use any message that they want for mapping while at the same time, JunctionBox automatically adds parameters to messages based on the interaction for that message. This achieves a balance between tolerance for customized messages while still providing basic functionality. Mapping features in JunctionBox are described in Chapter 4, Section 4.2, Subsection 4.2.2.

7.3 Networking Tolerance

Networking tolerance means that a toolkit should enable a range of networking options. While allowing networking, a toolkit should not get in the way of how that network is configured. Interfaces should be able to talk to an audio engine on the same device, in the same room over a wireless network, or over the internet. An interaction mapping toolkit should also allow interactions to be sent over the network to different computers.

That is, any interactive part of an interface should be able to be individually mapped to any computer. In JunctionBox, an interface can be designed that sends to a single other computer or different interactions on the interface can send messages to different computers. This opens up a a wider range of networking options, showing networking tolerance. Networking options in JunctionBox are described in Chapter 4, Section 4.2.

7.4 Graphical Tolerance

Graphical tolerance refers to the visual design options that a toolkit allows. An interaction toolkit should work well in a graphical context, allowing for visual feedback from any kind of interaction. At the same time, an interaction toolkit should minimize its constraints on graphical output. This allows for much greater freedom in designing the look of an interactive instrument. Other than some constraints on the shapes of interactive objects on an interface, JunctionBox makes no specification for how these objects appear visually. Graphical output in JunctionBox is described in Chapter 4, Section 4.2. Chapter 6 shows just a few of the visual possibilities enabled by graphical tolerance in JunctionBox.

7.5 Summary

In this chapter, I presented my design principles derived from the development and use of the JunctionBox toolkit. The most important factor for my research is the concept of interaction tolerance since it realtes directly to the application of my *unit interaction* model. The basic concept of interaction tolerance is the policy of non-inference. Non-interference, inherently supports the idea that interactions that can be discrete (hence unit) and can be combined to introduce chosen complexity levels. I also identified mapping, networking, and graphical tolerances as important principles in the design and

implementation of JunctionBox. These design principles distilled from my research offer guidance for toolkits that combine multi-touch interaction mapping, networking, and graphics for building musical interfaces.

Chapter 8

Conclusions

My training is as an engineer, and I consider that I'm not a composer, I'm not a professional performer of any instrument. I do love music. If I've done anything, I am an inventor of new instruments, and almost all the instruments I have invented are computer programs.

–**Max Mathews** [82]

In this thesis, I have described my investigation into the development and application of a *unit interaction* model to multi-touch, networked interactions for mapping to musical control. My research was motivated by my interest in creative coding tools that give developers the freedom to create highly-customized multi-touch interfaces that meet their specific needs and offer a range of possibilities from the simple to the complex. To instantiate this model, I designed and built JunctionBox, a creative coding toolkit for building multi-touch interfaces. To put the model into the context of current technologies and to test it in real musical situations, I compared JunctionBox to similar multi-touch mapping tools and used JunctionBox to build a series of musical interfaces that I used in my own performance practice.

My research in applying my *unit interaction* model via JunctionBox led to a number of contributions. In this chapter, I describe those research contributions along with some ideas for future research. The first section of this chapter details the contributions (8.1). This is followed by some of my ideas for future work that builds on this research (8.2). After that, I offer some closing remarks (8.3). The chapter ends with a short coda (8.4).

8.1 Contributions

During the course of my research, I answered a number of challenges as described in (Chapter 1, Section 1.5 and those answers are the contributions made by my research.

1. **The Development of a Unit Interaction Model for Multi-touch Interactions**

   By studying existing multi-touch instruments and mapping tools, I derived a set of unit mappable multi-touch interactions. I further developed the model during the course of building the JunctionBox toolkit. The goal for the model was to determine the most low-level multi-touch interactions that could be individually mapped to musical control. To find the most low-level interactions, I investigated two broad categories of interactions: 1) manipulating shapes with multi-touch input and 2) tracking one or more touches directly. Within these two categories, I found a rich set of unit interactions to use for mapping to musical control. As a thorough investigation of the possibilities inherent in multi-touch, the model serves as a contribution to the advancement of multi-touch input for music.

2. **The Creation of a Toolkit that Reifies the Unit Interaction Model**

   To test my unit interaction model, I reified the model by building the JunctionBox software toolkit. The unit interactions in JunctionBox can be used in various combinations for building multi-touch musical interfaces with varying degrees of complexity. The full set of unit interactions is described in Chapter 4, Section 4.3. In order to evaluate the success of my unit interaction model, with JunctionBox, I took two paths: a) a comparative analysis to show that my investigation into the unit interaction model yielded more mappable interactions than other mapping tools (Chapter 5) and b) that my unit interaction model is usable in my own musical practice (Chapter 6).

(a) Having identified the mappable multi-touch interactions in my unit interaction model, the goal was to make JunctionBox a interaction superset with a greater number of unit interactions than those offered by similar mapping tools. JunctionBox was compared to Control, TouchOSC, and Lemur in Chapter 5, showing that it offers significantly more mappable interactions than those tools. Going beyond unit interactions, I built special features into JunctionBox that no other tool offers, including saving interaction states, inheritable interactions, recording interactions, and making network and mapping setup easier. The special features are described in Chapter 4, Section 4.4.

(b) To show that my unit interaction model was usable in practice, I used JunctionBox to design and build a series of musical interfaces that I used in performances. The interfaces I designed show that JunctionBox can be used to build creative interfaces with a diversity of interactions and graphical styles. By building these interfaces and using them in actual performance situations, I showed that JunctionBox is performable software, ready for use in real musical situations. Chapter 6 describes the interfaces, how they were used for musical control, and when and where they were performed.

JunctionBox is itself a contribution, both as a reification of my unit interaction model and as a working toolkit for performing. By making JunctionBox open-sourced and free to download [33], I am giving the software back to the creative coding community to use for building multi-touch interfaces or to use as a reference implementation for creating their own multi-touch toolkit or for any other use.

3. **A Set of Design Principles**

My research into the design and development of JunctionBox led me to derive a set of design principles that distill the lessons I learned in making a toolkit that provides

functionality while allowing for creative freedom. The design principles are all variations on the same theme: tolerance. The particular tolerance principles that I derived include interaction tolerance, mapping tolerance, networking tolerance, and graphical tolerance. My design principles are significant contribution to knowledge in that they can be applied to the design and development of mapping toolkits that balance functionality with creativity. Chapter 7 describes the design principles that I derived from my research on the development of JunctionBox.

8.2 Future Work

The development of JunctionBox will continue beyond the research presented in this thesis. Here are the specific features that I want to add in future work:

- Input beyond multi-touch

  Multi-touch is such a common way to interact with computers that it is important to support it. However, phone and tablet devices contain a variety of sensors that could be mapped in JunctionBox. Some sensors that can be mapped are accelerometers, light sensors, and temperature sensors to name a few. In order to handle these sensors, I want to build a generic input framework into JunctionBox that treats all input as mappable to OSC messages and therefore to musical control.

- A new OSC library

  JunctionBox, as currently implemented, uses the JavaOSC library to create, send, and receive OSC messages. While JavaOSC works for this task, it does not have all of the features I would like it to have for use with JunctionBox. Some features I want in my OSC library include more strongly typed parameters, the ability to easily add parameters to multiple messages, and the ability to reference messages by name

rather than as object, and an option for message redundancy in unstable networking situations. All but the last of these are already implemented in JunctionBox. I would like to take that code and make it available outside of JunctionBox for others to use as a separate OSC library.

- Further development of NDEF

  NDEF is an idea that has, so far, only been used in testing situations to ensure that the basic functionality works in JunctionBox. While the specification works well in testing, I would like to begin to use NDEF in real situations. This means that I need to develop a generalized JunctionBox-based application that takes full advantage of NDEF. Another important step that would help to advance NDEF would be to work on downloadable NDEF receivers for a variety of audio engines.

- Interface building without coding

  While I am a great believer in the power of using code to build customized interfaces, I recognize that not everyone may have time to learn to code. Acknowledging this, I think that interface building through purely visual/interactive means is entirely possible with JunctionBox. This would open up my research work to a potentially larger audience.

- Release JunctionBox 1.0

  Once I have added all of the features that I want in JunctionBox, I will begin testing them with the goal of releasing a 1.0 version of the software. JunctionBox 1.0 will be open-sourced and available as a download for anyone interested in building highly customized multi-touch music controllers.

8.3 Closing Remarks

This thesis has detailed my research into the development of a *unit interaction* model and the use of that model in the design and implementation of the JunctionBox toolkit. The model addresses the challenge of offering greater freedom and creativity in building multi-touch musical interfaces by offering a large number of mappable unit interactions. I showed that JunctionBox has a greater number of mappable interactions than similar mapping tools, opening up the creative space of interaction design. Building my own interfaces with JunctionBox and using them in performances showed that my research has allowed more creative options for interfaces and that JunctionBox is usable as a working toolkit for building musical instruments.

Beyond JunctionBox, it is my hope that the methodology employed for my research can be employed in a variety of settings, even the development of toolkits that are far removed from the building of musical interfaces. Developing software, can, with the right rigor and methodology, be a valuable area of research. Researchers have an opportunity to both create something usable by their community and to share invaluable knowledge about design and development. My hope if that JunctionBox will not only be used by others to create musical interfaces but that it will inspire researchers to make a contribution by creating more software libraries and toolkits that take creative freedom into account.

8.4 Coda

I did not really know Max Mathews but he was at the Center for Computer Research in Music and Acoustics (CCRMA) during the time that I was working on a Master's degree there from 2007–2008. After finishing my degree at CCRMA, I worked there for a few months. One day, I made a mistake and managed to give myself a serious cut on my

finger (the scar is still there). So I ran to the lab where the first aid kit was to get myself a band-aid. Max was in the lab (the Max lab!) working on something at his desk. I went through the first aid kit and soon figured out that there were no band-aids to be found there. Max asked me what I was looking for and I told him that I needed a band-aid for a cut on my finger. He reached into his pocket, pulled out a band-aid, and offered it to me. I thanked him, put the band-aid on my finger, and went back to work.

In the quote that opens this chapter, Max downplays his own significance to the history of computer music. He didn't just write instruments that were computer programs, he wrote the very first music programs. Anyone, myself included, who builds instruments with software programs owes a tremendous debt to Max's pioneering work. As a non-composer, non-professional musician, I want to acknowledge the significant work that Max did that allows myself and others who love music to make a contribution, however small by comparison it may be.

Thanks for getting computer music started Max. And thanks for the band-aid.

Bibliography

[1] D. W. Bernstein. 'Listening to the Sounds of the People': Frederic Rzewski and Musica Elettronica Viva (1966–1972). *Contemporary Music Review*, 29(6):535–550, 2010.

[2] J. Bischoff, R. Gold, and J. Horton. Music for an Interactive Network of Microcomputers. *Computer Music Journal*, 2(3):24–29, 1978.

[3] T. Blaine. Home of the Jamodrum. `http://www.jamodrum.net/`, 2014.

[4] T. Blaine and C. Forlines. Jam-O-World: Evolution of the Jam-O-Drum Multiplayer Musical Controller into the Jam-O-Whirl Gaming Interface. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 1–6, 2002.

[5] T. Blaine and T. Perkis. The jam-o-drum interactive music system: a study in interaction design. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '00, pages 165–173, New York, NY, USA, 2000. ACM.

[6] E. Brandt and R. B. Dannenberg. Time in Distributed Real-Time Systems. In *Proceedings of International Computer Music Conference*, 1999.

[7] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. `http://tools.ietf.org/html/rfc7159`, 2014.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml/`, 2015.

[9] Brookhaven National Laboratory. Oldest Musical Instrument Dated. `http://www.bnl.gov/bnlweb/pubaf/pr/1999/bnlpr092299.html`, 1999.

[10] C. Brown and J. Bischoff. Early Concerts of the League. http://crossfade.walkerart.org/brownbischoff/league_texts/early_league_concerts_f.html, 2002.

[11] C. Brown and J. Bischoff. Indigenous to the Net. http://crossfade.walkerart.org/brownbischoff/introduction_main.html, 2002.

[12] B. Buxton. Artists and the Art of the luthier. *SIGGRAPH Computer Graphics*, 31(1):10–11, February 1997.

[13] V. Cerf, Y. Dalal, and C. Sunshine. Specification of internet transmission control program. http://tools.ietf.org/html/rfc675, 1974.

[14] C. Chafe, M. Gurevich, G. Leslie, and S. Tyan. Effect of Time Delay on Ensemble Accuracy. In *Proceedings of the International Symposium on Musical Acoustics*, 2004.

[15] J. M. Chowning. The Synthesis of Complex Audio Spectra by Means of Frequency Modulation. *Computer Music Journal*, 1(2):46–54, 1977.

[16] ChucK Team. ChucK. http://chuck.cs.princeton.edu/, 2015.

[17] ChucK Team. ChucK. http://chuck.cs.princeton.edu/doc/program/ugen_full.html#BandedWG, 2015.

[18] P. Community. Pure Data. http://puredata.info/, 2014.

[19] Computer History Museum. IBM 704 Electronic Data Processing System. http://www.computerhistory.org/revolution/early-computer-companies/5/113/489.

[20] P. Cook. Principles for designing computer music controllers. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 1–4, 2001.

[21] A. Curran. MEV letter. http://www.alvincurran.com/writings/mev.html, 1989.

[22] Cycline 74. Mira. https://cycling74.com/products/mira/, 2015.

[23] Cycling 74. Max is a visual programming language for media. http://cycling74.com/products/max/.

[24] Cycling 74. Max/MSP. https://cycling74.com/products/max/, 2015.

[25] J.-P. Cáceres and A. B. Renaud. Playing the network: the use of time delays as musical devices. In *Proceedings of International Computer Music Conference*, pages 244–250, 2008.

[26] R. B. Dannenberg. Impressions from the SMC Roadmap. *Journal of New Music Research*, 36(3):191–196, 2007.

[27] P. L. Davidson and J. Y. Han. Synthesis and control on large scale multi-touch sensing displays. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 216–219, 2006.

[28] S. Developers. SuperCollider. http://supercollider.github.io/, 2014.

[29] R. A. Diaz-Marino, E. Tse, and S. Greenberg. Programming for Multiple Touches and Multiple Users: A Toolkit for the DiamondTouch Hardware. In *UIST 2003 Conference Companion*, 2003.

[30] Dictionary.com. Junction. http://dictionary.reference.com/browse/junction?s=t, 2015.

[31] Dictionary.com. Tolerate. http://dictionary.reference.com/browse/tolerate?s=t, 2015.

[32] P. Dietz and D. Leigh. DiamondTouch: A Multi-user Touch Technology. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 219–226, New York, NY, USA, 2001. ACM.

[33] L. Fyfe. JunctionBox. `http://innovis.cpsc.ucalgary.ca/Software/JunctionBox`, 2015.

[34] L. Fyfe, A. Pon, and S. Fay. Aspect. `http://innovis.cpsc.ucalgary.ca/Research/Aspect`, 2015.

[35] L. Fyfe, A. Tindale, and S. Carpendale. JunctionBox for Android: An Interaction Toolkit for Android-based Mobile Devices. In *Proceedings of the Linux Audio Conference*, pages 89–92, 2012.

[36] L. Fyfe, A. Tindale, and S. Carpendale. Node and Message Management with the JunctionBox Interaction Toolkit. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 520–521, 2012.

[37] L. Fyfe, A. Tindale, and S. Carpendale. Extending the Nexus Data Exchange Format (NDEF) Specification. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 343–346, 2014.

[38] A. Gokcezade, J. Leitner, and M. Haller. LightTracker: An Open-Source Multitouch Toolkit. *ACM Computers in Entertainment*, 8(3):19:1–19:16, December 2010.

[39] Gorillaz. The Fall. `http://thefall.gorillaz.com/`, 2010.

[40] S. Greenberg and B. Buxton. Usability Evaluation Considered Harmful (Some of the Time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 111–120, 2008.

[41] S. Gresham-Lancaster. The Aesthetics and History of the Hub: The Effects of Changing Technology on Network Computer Music. *Leonardo Music Journal*, 8:39–44, 1998.

[42] M. Gurevich. JamSpace: Designing A Collaborative Networked Music Space for Novices. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 118–123, 2006.

[43] M. Gurevich, C. Chafe, G. Leslie, and S. Tyan. Simulation of Networked Ensemble Performance with Varying Time Delays: Characterization of Ensemble Accuracy. In *Proceedings of the International Computer Music Conference*, 2004.

[44] G. Hajdu. Quintet.net–A Quintet on the Internet. In *Proceedings of the International Computer Music Conference*, pages 315–318, 2003.

[45] J. Y. Han. Low-Cost Multi-Touch Sensing through Frustrated Total Internal Refl ection. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 115–118, 2005.

[46] T. E. Hansen, J. P. Hourcade, M. Virbel, S. Patali, and T. Serra. PyMT: a post-WIMP multi-touch user interface toolkit. In *Proceedings of the Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 17–24, New York, NY, USA, 2009. ACM.

[47] hexler.net. TouchOSC. `http://hexler.net/software/touchosc`, 2015.

[48] hexler.net. TouchOSC Control Reference. `http://hexler.net/docs/touchosc-controls-reference`, 2015.

[49] A. Hunt, M. M. Wanderley, and M. Paradis. The importance of parameter mapping in electronic instrument design. In *Proceedings of the Conference on New Interfaces*

for Musical Expression, pages 1–6, 2002.

[50] T. Ichiyanagi. Distance, 1961.

[51] JazzMutant. Lemur. `http://www.jazzmutant.com/lemur_gallery_pics.php`, 2014.

[52] JazzMutant. Lemur. `http://www.jazzmutant.com/`, 2014.

[53] S. Jordà. Faust music on line: An approach to Real-Time collective composition on the internet. *Leonardo Music Journal*, 9:5–12, 1999.

[54] S. Jordà. Sonigraphical Instruments: From FMOL to the reacTable*. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 70–76, 2003.

[55] S. Jordà. Instruments and players: Some thoughts on digital lutherie. *Journal of New Music Research*, 33(3):321–341, 2004.

[56] S. Jordà, M. Kaltenbrunner, G. Geiger, and R. Bencina. The reactable. In *Proceedings of the International Computer Music Conference*, pages 579–582, 2005.

[57] M. Kaltenbrunner and R. Bencina. reacTIVision: A Computer-vision Framework for Table-based Tangible Interaction. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, pages 69–74, New York, NY, USA, 2007. ACM.

[58] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO - A Protocol for Table-Top Tangible User Interfaces. In *Proceedings of the International Workshop on Gesture in Human-Computer Interaction and Simulation*, 2005.

[59] D. Kammer, M. Keck, G. Freitag, and M. Wacker. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. In *Workshop on Engineering Patterns for Multitouch Interfaces*, 2010.

[60] N. P. Lago and F. Kon. The quest for low latency. In *Proceedings of the International Computer Music Conference*, pages 33–36, 2004.

[61] C. Latta. Notes from the NetJam Project. *Leonardo Music Journal*, 1(1):103–105, 1991.

[62] U. Laufs, C. Ruff, and J. Zibuschka. MT4j – A Cross-platform Multi-touch Development Framework. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2010.

[63] S. Lee, W. Buxton, and K. Smith. A multi-touch three dimensional touch-sensitive tablet. *ACM SIGCHI Bulletin*, 16(4):21–25, 1985.

[64] Liine. Lemur. `https://liine.net/en/products/lemur`, 2014.

[65] Liine. Lemur dj template. `https://liine.net/en/community/user-library/view/118/`, 2015.

[66] Liine. Lemur User Guide. `https://liine.net/assets/files/lemur/Lemur-User-Guide.pdf`, 2015.

[67] Liine. Objects. `https://liine.net/en/products/lemur/objects`, 2015.

[68] G. Loy. Musicians Make a Standard: The MIDI Phenomenon. *Computer Music Journal*, pages 8–26, 1985.

[69] J. Lyst. tuiozones. `http://jlyst.com/tz/`, January 2011.

[70] J. Malloch, S. Sinclair, and M. M. Wanderley. Libmapper: (a Library for Connecting Things). In *Proceedings of the International Conference on Human Factors in Computing Systems*, pages 3087–3090, 2013.

[71] M. V. Mathews. The Digital Computer as a Musical Instrument. *Sciece*, 142(3592):553–557, 1963.

[72] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[73] MIDI Manufacturers Association and Others. *The complete MIDI 1.0 detailed specification: incorporating all recommended practices*. MIDI Manufacturers Association, 1996.

[74] R. Mills. Dislocated sound: A survey of improvisation in networked audio platforms. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 186–191, 2010.

[75] NETescopio. FMOL. `http://netescopio.meiac.es/en/obra.php?id=155`, 2015.

[76] P. A. Nilsson. Control or Play? In *Proceedings of the Electroacoustic Music Studies Network Conference*, 2014.

[77] NUI Group. Diffused Illumination (DI). `http://wiki.nuigroup.com/Diffused_Illumination`, 2015.

[78] NUI Group. Frustrated Total Internal Reflection (FTIR). `http://wiki.nuigroup.com/FTIR`, 2015.

[79] M. Nyman. *Experimental music: Cage and beyond*. Cambridge University Press, 2 edition, 1999.

[80] P. Olivo and N. Roussel. Boing - A flexible multi-touch toolkit. `https://boing.readthedocs.org/en/latest/`, June 2014.

[81] E. Paluka, K. K. Sikes, Z. Cook, C. Collins, and M. Hancock. Simple Multi-Touch Toolkit. `http://vialab.science.uoit.ca/smt/`, June 2014.

[82] T. H. Park. An Interview with Max Mathews. *Computer Music Journal*, 33(3):9–22, 2009.

[83] J. Patten. Audiopad. `http://www.jamespatten.com/audiopad/`, 2014.

[84] J. Patten, H. Ishii, J. Hines, and G. Pangaro. A Wireless Object Tracking Platform for Tangible User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 253–260, 2001.

[85] J. Patten, B. Recht, and H. Ishii. Audiopad: a tag-based interface for musical performance. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 1–6, 2002.

[86] J. Postel. User Datagram Protocol. `http://tools.ietf.org/html/rfc768`, 1980.

[87] J. Postel. Internet Control Message Protocol. `http://tools.ietf.org/html/rfc792`, 1981.

[88] J. Postel. Internet Protocol. `http://tools.ietf.org/html/rfc791`, 1981.

[89] M. Puckette. Pure data: another integrated computer music environment. In *Proceedings of the International Computer Music Conference*, pages 37–41, 1996.

[90] Reactable. Reactable - the electronic music instrument, the Music App and DJ tool. `http://reactable.com/`, 2014.

[91] Reactable. Reactable mobile Manual. http://reactable.com/mobile/manual/gestures.html, July 2014.

[92] reacTIVision. a toolkit for tangible multi-touch surfaces. http://reactivision.sourceforge.net/, 2014.

[93] C. Reas and B. Fry. Processing: programming for the media arts. *AI & Society*, 20(4):526–538, 2006.

[94] P. Rebelo and A. B. Renaud. The frequencyliator: distributing structures for networked laptop improvisation. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 53–56, 2006.

[95] A. B. Renaud, A. Carôt, and P. Rebelo. Networked Music Performance: State of the Art. In *Proceedings of the AES 30th International Conference*, 2007.

[96] C. Roads. *The Computer Music Tutorial*. MIT press, 1996.

[97] C. Roads and M. Mathews. Interview with Max Mathews. *Computer Music Journal*, 4(4):15–22, 1980.

[98] Robert Andrews. Reactable tactile synth catches björk's eye – and ear. http://archive.wired.com/entertainment/music/news/2007/08/bjork_reacTable, 2007.

[99] C. Roberts. Control. http://charlie-roberts.com/Control/, August 2014.

[100] C. Roberts, G. Wakefield, and M. Wright. Mobile Controls On-The-Fly: An Abstraction for Distributed NIMEs. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 2012.

[101] F. Rzewski and M. Verken. Musica Elettronica Viva. *The Drama Review: TDR*, 14(1):92–97, 1969.

[102] P. Schaeffer. Acousmatics. In Christoph Cox and Daniel Warner, editor, *Audio Culture: Readings in Modern Music*. Continuum, 2004.

[103] A. W. Schmeder and M. Wright. A Query System for Open Sound Control. In *Proceedings of the Open Sound Control Conference*, 2004.

[104] S. Smallwood, D. Trueman, P. R. Cook, and G. Wang. Composing for Laptop Orchestra. *Computer Music Journal*, 32(1):9–25, 2008.

[105] J. Stelkens. peerSynth: A P2P Multi-User Software Synthesizer with new techniques for integrating latency in real time collaboration. In *Proceedings of the International Computer Music Conference*, pages 319–322, 2003.

[106] G. Sullivan. Research Acts in Art Practice. *Studies in Art Education*, 48(1):19–35, 2006.

[107] A. Tanaka. Sensorband (1993–2003). `http://cec.sonus.ca/econtact/14_2/tanaka_gallery.html`, 2012.

[108] A. Tanaka. The Use of Electromyogram Signals (EMG) in Musical Performance. *eContact!*, 14(2), 2012.

[109] A. Tanaka. Sensorband. `http://www.ataut.net/site/Sensorband`, 2015.

[110] S. Tarakajian, D. Zicarelli, and J. K. Clayton. Mira: Liveness in iPad controllers for Max/MSP. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 421–426, 2013.

[111] The Processing Foundation. Processing. `https://processing.org/`, 2015.

[112] D. Trueman. Why a laptop orchestra? *Organised Sound*, 12(02):171–179, 2007.

[113] D. Trueman, P. Cook, S. Smallwood, and G. Wang. PLOrk: the Princeton Laptop Orchestra, Year 1. In *Proceedings of the International Computer Music Conference*, pages 443–450, 2006.

[114] E. Varèse. The Liberation of Sound. In Christoph Cox and Daniel Warner, editor, *Audio Culture: Readings in Modern Music*. Continuum, 2004.

[115] G. Wang. Stanford Laptop Orchestra (SLOrk). `http://slork.stanford.edu/`, 2015.

[116] G. Wang, N. Bryan, J. Oh, and R. Hamilton. Stanford Laptop Orchestra (SLOrk). In *Proceedings of the International Computer Music Conference*, 2009.

[117] G. Wang and P. Cook. ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the ACM International Conference on Multimedia*, pages 812–815, New York, NY, USA, 2004. ACM.

[118] G. Weinberg. The Aesthetics, History, and Future Challenges of Interconnected Music Networks. In *Proceedings of the International Computer Music Conference*, pages 349–356, 2002.

[119] G. Weinberg. Interconnected Musical Networks: Toward a Theoretical Framework. *Computer Music Journal*, 29(2):23–39, June 2005.

[120] E. W. Weisstein. Golden Ratio. `http://mathworld.wolfram.com/GoldenRatio.html`, 2015.

[121] D. Wessel and M. Wright. Problems and prospects for intimate musical control of computers. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 1–4, 2001.

[122] Wikipedia. Kim-1. `http://en.wikipedia.org/wiki/KIM-1`, 2015.

[123] Wikipedia. Orrery. http://en.wikipedia.org/wiki/Orrery, 2015.

[124] M. Wright. The Open Sound Control 1.0 Specification. http://opensoundcontrol.org/spec-1_0, 2002.

[125] M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

[126] M. Wright and A. Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the International Computer Music Conference*, pages 101–104, 1997.

[127] M. Wright, A. Freed, A. Lee, T. Madden, and A. Momeni. Managing Complexity with Explicit Mapping of Gestures to Sound Control with OSC. In *Proceedings of the International Computer Music Conference*, pages 314–317, 2001.

[128] Álvaro Barbosa. Displaced Soundscapes: A Survey of Network Systems for Music and Sonic Art Creation. *Leonardo Music Journal*, 13:53–59, 2003.

[129] Álvaro Barbosa. Public Sound Objects: a shared environment for networked music practice on the Web. *Organised Sound*, 10(3):233, 2005.

Appendix A

Code Examples

The JunctionBox software package contains code examples that demonstrate the mappable interactions described in Chapter 4, Section 4.3. The examples are divided into three categories depending on the source of input: Android, Mouse, and TUIO. Within each category, the examples feature a specific type of interaction with the name of the example representing the interaction or set of interactions. The following sections feature a screenshot of each example with a brief description of what it does.

A.1 Example 1: Activation and Toggling

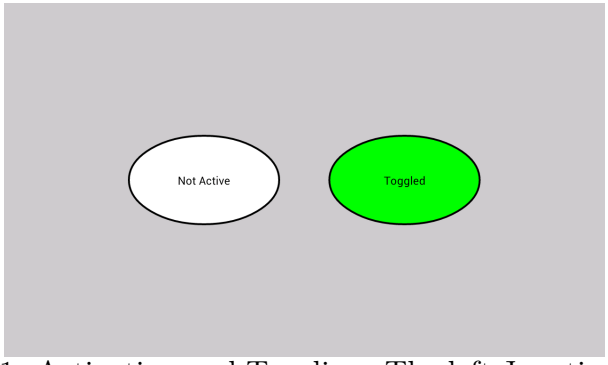


Figure A.1: Example 1: Activation and Toggling. The left Junction is not active and the right Junction is toggled.

This example demonstrates the activation and toggling interactions. The two are included in the same example because they are similar enough that there was no need to have separate examples. The Android version, shown in Figure A.1 features two ovals, one to be activated and one to be toggled.

A.2 Example 2: Translation

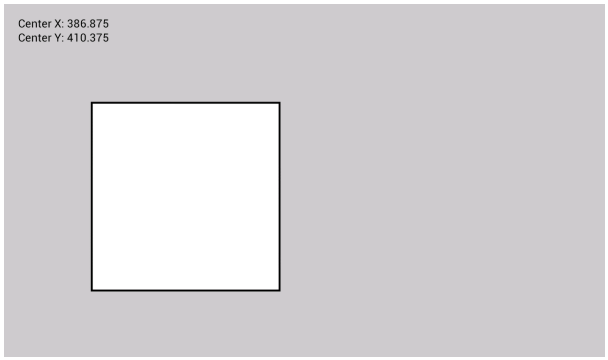


Figure A.2: Example 2: Translation. A Junction after it has been translated. Note the Junctions XY center in the upper left corner.

The translation example is a square that can be moved anywhere on the screen. The coordinates for the center of the square, relative to the size of the screen, are shown in the upper left corner. They change as the square is translated across the screen.

A.3 Example 3: Rotation

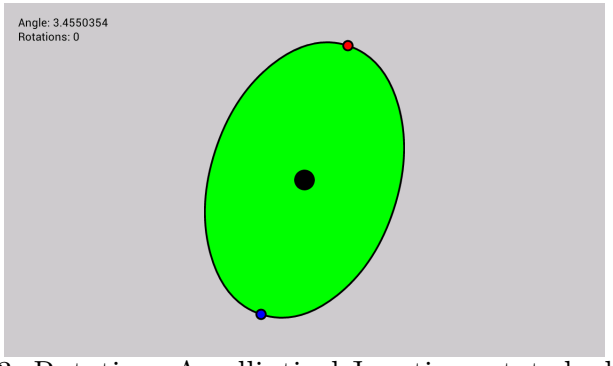


Figure A.3: Example 3: Rotation. An elliptical Junction rotated a little over 180 degrees.

In the rotation example, an oval is rotated with two small red and blue circles to better show the change in angle. Both the current angle and the number of rotations the oval has undergone are shown in the upper left corner.

A.4 Example 4: Scaling

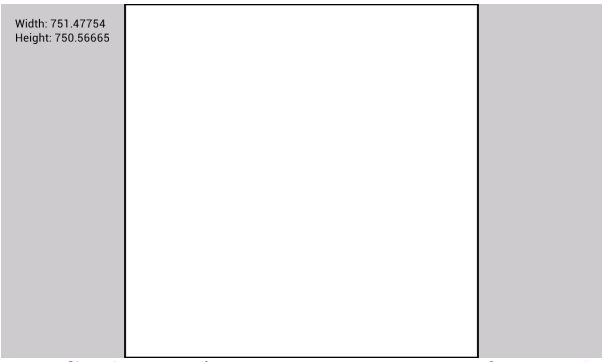


Figure A.4: Example 4: Scaling. A square Junction after it has been scaled to approximately its maximum height. Note the width and height values in the upper left corner.

The scaling example features a square that can be scaled with two touches. In the upper left corner of the example, is the current size of the square in terms of the size of the screen. Note that in Figure A.4, the square has reached its maximum size since width and height are scaled proportionally and the height of the screen is less than the width.

A.5 Example 5: Touches

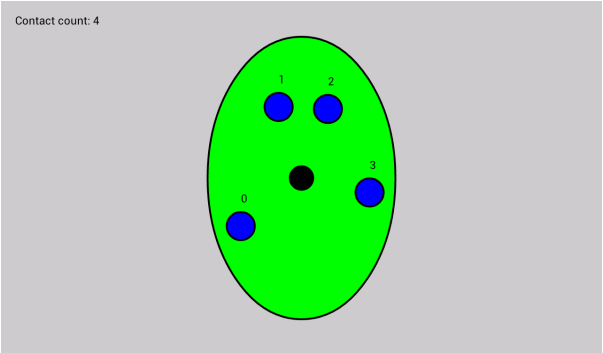


Figure A.5: Example 5: Touches. An ovular Junction with four touches represented by blue dots. Each touch/blue dot has and identifier. Note the touch count shown in the upper left corner.

For the touches example, an oval will track as many touches as the device will allow.

The device that I used to show this example only allows four touches. Note that the ID for each touch is shown above the blue dot that represents the touch. The number of active touches is shown in the upper left corner.

A.6 Example 6: Saving

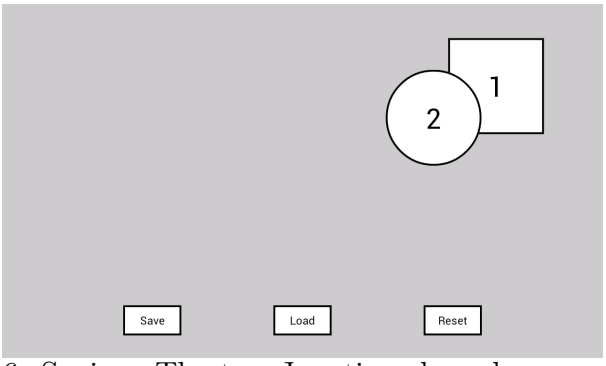


Figure A.6: Example 6: Saving. The two Junctions have been moved. The save button will save their current location among other values. The load button will load the saved values. The reset button moves the Junctions back to their original locations.

The saving example has two translatable Junctions. After moving the Junctions around, their location can be saved with the "Save" button. Using the "Reset" button puts the Junctions back to their original position. The "Load" button can then be used to put the Junctions back into their saved position. Note that the two Junctions can overlap but that they can also change their order such that selecting the Junction on the bottom will bring it to the top.

A.7 Example 7: Inheriting

For the inheriting example, three Junctions of different sizes overlap. Rotating the outermost Junction rotates the inner two. Rotating the middle Junction rotates the smallest Junction. Not that the Junctions inherit relative angle changes. This means that if one of the inner Junctions is rotated and then its outer parent Junction is rotated, the inner

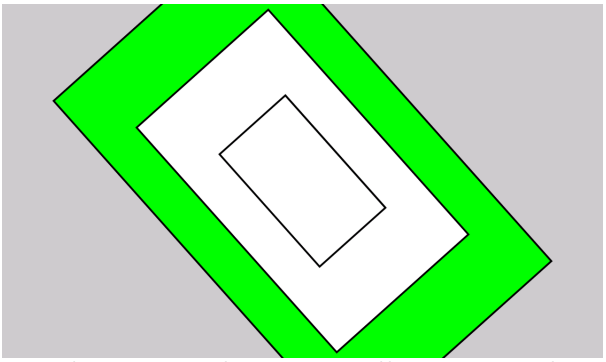


Figure A.7: Example 7: Inheriting. The two smaller rectangular Junctions inherit their angles from the parent Junction.

Junction adds to its current angle. This is opposed to having the inner Junctions always match the exact angle of the outer Junction.

A.8 Example 8: Recording

The recording example is more complex than the previous examples since it combines many interactions into a single interface. There are recording, playing, looping, stopping, clearing, stretching, saving, and loading buttons along the left side. The large yellow rectangle is the area for recording interactions. The interaction involves touches moving inside of the recording area, drawing a black line as they move. Once recorded, the interaction, including the black line are played back with the correct location of the touches and the correct timing for the interaction. The slider at the bottom can be used to change the timing of the recorded interaction by moving to the left for faster times and moving to the right for slower times. Once new timing is selected with the slider, the brown stretch button on the left sets the interaction to the new timing. In Figure A.8, an interaction is being recorded.

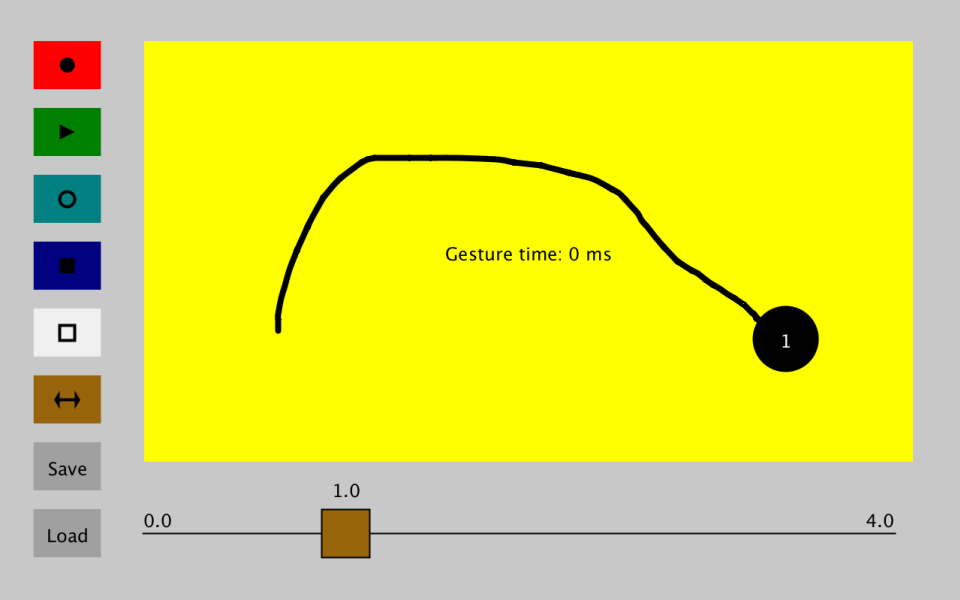


Figure A.8: Example 8: Recording. The yellow square is a recording area. Notice that the record button is brighter, indicating that recording is happening. Since recording is not finished, the time for the interaction in the recording area shows 0 milliseconds.

Appendix B

The NDEF Specification

The Nexus Data Exchange Format (NDEF) is an Open Sound Control (OSC) namespace specification that, when implemented, makes connection and message management tasks easier for developers of OSC-based systems. The NDEF specification can be implemented by any system that can handle standard OSC messages.

B.1 Connection Management

One of the central features of NDEF is node identification. All NDEF messages have the IP address and port of the message source (OSC server or client) as the first two arguments. The generalized form of NDEF messages is:

```
/ndef/[container]/[method] [IP address] [port]
```

This kind of identification is particularly useful for cases where one OSC client is in a one-to-many relationship with multiple OSC servers or where multiple OSC clients are in a many-to-many relationship with multiple OSC servers.

To begin an NDEF exchange, the OSC client sends out a request message. The type tag for the request is si where the IP address is the string and the port is the integer.

```
/ndef/connection/request,si
```

The NDEF exchange system is similar in some ways to the TCP handshake [13] in which both ends acknowledge the connection. However, NDEF is a two way exchange rather than a three-way handshake. When a connection request is received (and accepted), an accept message is sent to the OSC client.

```
/ndef/connection/accept,si
```

This exchange allow nodes to determine whether another node has NDEF capabilities and whether it is available for connections. Another important element of the exchange is that both nodes then can identify each other using IP address and port as unique identifiers. When multiple nodes may be involved in a network, this identification is essential.

In creating an NDEF connection, nodes identify each other by their source IP address and port number. Then all further message exchanges can be identified as belonging to a specific connection. Once a connection is established, it is useful to be able to test the connection. In order to allow for connection testing, the namespace now has a "ping" message:

```
/ndef/connection/ping,si
```

The string and integer arguments are the IP address and port of the message sender.

If the node receiving the "ping" message is available, it will send an "echo" message to the originating node:

```
/ndef/connection/echo,si
```

The ping/echo exchange can be used to determine both availability and round-trip time (latency). NDEF implementations are free to determine how frequently "ping" messages are sent and what sort of time-out mechanism is in place when "echo" messages have not been received.

NDEF ping messaging does not use the ICMP protocol [87] used by the ping utility available with most operating systems. While an ICMP ping is useful for determining the general availability of a node on the network, it cannot determine when both OSC

messaging and NDEF messaging are available. A node might be available on the network with neither essential service running.

The ping/echo exchange would be useful, for example, in a laptop orchestra. Often it is not always clear that nodes (laptops) in the orchestra are actually receiving the intended OSC messages. With the NDEF ping/echo, laptops could be periodically polled to determine whether they are actually receiving OSC messages. This polling makes it easier to address issues with any nodes that are not receiving messages.

B.2 Message Management

Once a connection has been established between two or more nodes, each node can send a message request to the other nodes. A message request has a form similar to a connection request.

```
/ndef/message/request,si
```

The reply message has a slightly different form than the other messages in the namespace since, besides the IP address and port of the message source, it has the OSC message encoded as a string as the last argument.

```
/ndef/message/reply,sis
```

An example reply message containing the OSC message /foo:

```
/ndef/message/reply, "192.168.1.1" 7000 "/foo"
```

Any number of reply messages can be received by a requesting node once an initial request has been sent. This allows for some flexibility in the setup of OSC servers including the sending of new message replies as they are created on the server. NDEF does not provide a message format for setting ranges on OSC arguments. The reason

for this is that JunctionBox values are always sent normalized from 0-1. This eliminates the need to set ranges since OSC servers will simply scale arguments to any range without the interface being concerned with the specific range. Since all numbers output by JunctionBox are normalized, all numbers are floats.

Using NDEF, nodes can exchange messages in their OSC namespace. This allows a node's OSC namespace to be modelled by other nodes. Since all NDEF exchanges carry identifying information, each node can be modelled with a distinct namespace.

The new message management NDEF extensions allow nodes to go beyond just having models of other node's namespaces. The new extensions allow nodes to add, remove and replace the messages of other nodes, allowing for namespace synchronization. By synchronizing namespaces with NDEF, the task of having OSC clients and servers share a common namespace is made easier.

The new "add" message allows a node to add a message to another node:

```
/ndef/message/add,sis
```

An example for adding would be the use of multiple "add" messages to build an entire namespace on another node. The messages in that namespace could then be mapped by that node.

Messages can also be removed from a node with the new "remove" message:

```
/ndef/message/remove,sis
```

The "remove" message would be useful for removing parts of a namespace that are no longer shared between the nodes.

The new add and remove extensions having the following basic structure:

```
/ndef/message/[method] [IP address] [port] [OSC message]
```

The OSC message argument should include the address pattern and the type tag. Arguments should not be included.

A message can be replaced with another message with the "replace" message:

```
/ndef/message/replace,siss
```

The replace extension has four arguments with the last two arguments being the old message string and the new message string that will replace it. The general structure of replace messages:

```
/ndef/message/[method] [IP address] [port] [old message] [new message]
```

Of the three new messages, the "replace" message is the most powerful since the fact that it replaces a message means that the mapping using that message should still be valid. For example, an OSC client with an existing namespace could replace a message on an OSC server. This could be done by an interface designed to simply switch two messages. No typing would be necessary. But now the OSC server would have a new message that retains the same mapping as the previous message without the OSC client having to know about the details of the mapping on the OSC server.

The "add", "remove", and "replace" messages are not just meant to be sent to OSC servers that contain mappings. They can also be sent from OSC servers to clients used for control. This allows for two-way namespace synchronization.

Appendix C

JunctionBox Revisions

This appendix contains a revision history for each version of JunctionBox. The version number for each revision is included along with the date of the release for taht revision. Each revision contains the following categories of change:

New Features

New features added to JunctionBox.

API Changes

Changes to the JunctionBox API.

Bug Fixes

A bug fix that does not change the API.

Internal Changes

Reorganization of code that has no effect on the API.

Android

Changes specific to the Android version of JunctionBox.

Within categories, changes to specific classes are noted with the name of the class. Table C.1 is a legend with the markings that denote the change type for categories or classes.

Table C.1: Revision Legend

+	Added feature
=	Change in code with no change in functionality
-	Removed code

JunctionBox 0.98 - 30 March 2015

API Changes

Distpatcher

= Moved all NDEF functions from the Dispatcher to the the new Nexus object.

\- Removed the constructors Dispatcher() and Dispatcher(String address, int port) since they did not require arguments for boxWidth and boxHeight. Without boxWidth and boxHeight, JunctionBox will simply not work and that would be silly.

\+ Added clearJunctions() to enable the removal of all Junctions.

= Changed startPlayback(), stopPlayback(), getPlaybackTime() to startPlaying(), stopPlaying(), and getPlayTime(). This is more consistant with the other methods like startRecording() and others. It also generally reads better.

\+ Changed saveXML() and loadXML() so that they take OutputStreams and InputStreams directly for writing files.

\- Removed readFile() and writeFile() since saveXML() and loadXML() now handle this.

Junction

\+ Added getTargetAddress() and getTargetPort().

+ The new allowSaving() and isSavable() methods are related to the new XML saving/loading methods in the Dispatcher. Junction data can only be saved it is allowed.

Relay

+ Can change the IP address and port number with the new setSocket() method.
+ Added the getMessageCount() method.
+ The new addFloat(String message, float f, float min, float max) method allows for specifying the message and mapping the value. This should have been there in an earlier release. Better late than never.

Action

= Changed Action.CONTACT_COUNT and Action.ROTATION_COUNT to Action.COUNT_CONTACTS and Action.COUNT_ROTATIONS. All Actions should be named with a verb first. They are actions after all.

+ Added new Action.CONTACT_THETA for sending the angle of a Contact relative to the center of an ellipse.

Nexus

+ To test NDEF connections, added isConnected(String ip, int port) and isConnected(Relay r) methods.
+ Changed all NDEF related methods to be of the form send[Message] to clarify their purpose. Each of these has overridden methods accepting

a single IP, a list of IPs, or no argument (for sending to the default target). The new methods are sendConnectionRequest(), sendMessageRequest(), sendMessageAdd(), sendMessageRemove(), and sendMessageReplace().

+ New getRelay(int r) and getRelayCount() methods return a specific Relay or the number of Relays respectively.

internal changes

Distpatcher

+ Added the setDelayValue() method to the Event inner class to enable the building of event queues from XML files.

= Changed addEvent(), updateEvent() and removeEvent() methods (including overridden ones) to queueAddEvent(), queueUpdateEvent(), and queueRemoveEvent(). This reads better.

= Moved code in the now defunct addEvent(), updateEvent(), and removeEvent() methods into addContact(), updateContact(), and removeContact() to avoid overriding methods for calls to queueAddEvent(), queueUpdateEvent(), and queueRemoveEvent().

- Removed getEventQueue() that returned a PriorityQueue. Better to use getEvents() that returns an array.

+ Added scaleEventTimes() that takes a double and multiples it by the delay time for each Event, thus scaling the time for all Events.

+ The saveXML() and loadXML() methods now call saveJunctions(), saveRelays(), saveEvents() and loadJunctions(), loadRelays(), and loadEvents(),

new methods for reading and writing data to XML to be output to a file.

Junction

= Changed code inside of Junction.updateContact() so that messages for Action.CONTACT will send different values based on whether the Junction is a rectangle or an ellipse. A rectangle will send x,y while an ellipse will send r,theta.

= Fixed Contact mapping code for ellipses that are not circles. It turns out that theta will not work to find a point on an ellipse. In order to normalize the r value, a point on the ellipse must be found. Now the trig is right and the values look nicely normalized.

Nexus

= Put a null check and an isListening() check for OscPortIn inside of stopListening(). Otherwise, a NullPointerException will issue forth.

+ The acceptMessage() method now has code to take and set labels for Relays from /ndef/connection/accept messages.

JunctionBox 0.97 - 18 July 2013

API Changes

+ Added Junction.allowRotation(boolean, int) where the int argument represents the number of Contacts used for rotation. Allowed values are 1 or 2 Contacts.

+ Added Action.ROTATE_1 and Action.ROTATE_2 to enable the mapping of separate messages for 1 and 2 Contact rotations.
+ New Junction.allowTranslation(boolean, int) method allows for an arbitrary (greater than 0) number of Contacts for translation.
+ Similarly, Junction.allowTranslation(boolean, int, int) allows for a range of Contacts to be set for translation. The range is inclusive.
= In Junction, changed setLive() to beLive() since it reads better.
= Changed Action.TRANSLATE_XY to Action.TRANSLATE to be more consistant with other Actions.
= Changed Action.CONTACT_XY to Action.CONTACT for the same reason.

Bug Fixes

= In Junction, added a check to mapMessage() and unmapMessage() for the existance of the targetRelay. This insures that if no IP address and port are specified, no NullPointerException will occur.

Internal Changes

+ Calling Junction.allowRotation(true) with no integer argument enables both 1 and 2 Contact rotations. These can also be separately controlled with Junction.allowRotation(boolean, boolean).
+ Junction.setAngle() now relays 1 and 2 Contact rotation mappings if they are set.
+ In Junctions, added new minTranslationContacts and maxTranslationContacts integers for setting the range of Contacts to be used for trans-

lation.

+ Added new contactCount integer to Junction.updateContact() for updating subjunctions.

+ The new ROTATE_1 and ROTATE_2 Actions are now checked inside of mapMessage() and unmapMessage().

= Replaced rotationContacts integer and rotateable boolean with rotatable1 and rotatable2 booleans. Added relay 1 and 2 booleans along along with the related lists. The rotatableN booleans are now checked instead of the rotationContacts integer.

= Removed various relay booleans from inside of Junction.unmapMessage() since they didn't need to be set in this method.

JunctionBox 0.96 - 30 May 2013

New Features

+ Added setLabel() and getLabel() to allow a String to be used as a label for Junctions and Relays.

+ In Relay, the add[argument] methods now add a new OSC message along with the argument if it is not contained in the Relay.

Dispatcher

+ Added new NDEF messages to the Dispatcher: "/ndef/message/add", "/ndef/message/remove", and "/ndef/message/replace". Each of these new messages has a corresponding method for sending. The Dispatcher can receive these messages as well.

+ New methods for controlling recording and playback include: startRecording(), stopRecording(), isRecording(), getRecordTime(), startPlayback(), stopPlayback(), isPlaying(), loopPlayback(), startLooping(), stopLooping(), and getPlaybackTime(). Since there is so much new code here, see the source for more information.
+ For event management, added: getEventCount() and clearEvents().
+ The Dispatcher can write event data to an XML file. New methods are saveXML(), loadXML(), readFile() and writeFile(). For this release, only event data can be written to XML.

Junction

+ Added allowRecording() and isRecordable() to control the recording of events. Use these to determine which Junctions the Dispatcher will record events for.
+ The allowRotation() and allowTranslation() methods now accept an integer that controls the number of Contacts used for rotation and translation respectively. Rotation can be done with 1 or 2 Contacts. Translation can be done with 1-3 Contacts.

Internal Changes

Distpatcher

+ For recording and playing back events, added the Event and Player private classes. Many other variables have been added for event recording and playback. See the source code for a full list.
+ Added an event queue to the Dispatcher to enable the recording of contact events.

+ For internal event management: addEvent(), updateEvent(), removeEvent(), and getEvents().
+ The initialize() method now creates the event queue as well as the XML document writer.
+ Added checks in addContact(), updateContact(), and removeContact() to determine whether to record events associated with Junctions that have recording enabled.

Junction

= When Contacts are mapped to messages, the Y value is now inverted with the top edge of a rectangle mapping to 0 and the bottom edge mapping to 1. This is more consistent with Y values in Processing.
+ Added code in updateContact() to enable 2 Contact rotation and 1-3 Contact translation. Added some class-level variables that are associated with this code. See the source code for more information.

Relay

= Simplified the code in resetMessage(String a).
= Fixed incorrect code in the replaceMessage(String a1, Strng a2) method.

JunctionBox 0.95 - 10 December 2012

First Public Release

New Features

Distpatcher

+ Added setTarget() method to allow for target setting after initialization.

+ Added addContact(), updateContact() and removeContact. The code in these methods existed in the related _TuioCursor() methods. The _TuioCursor() methods now call the new _Contact() methods but the _Contact() methods can also be called without invoking the _TuioCursor methods.
+ Added clearContacts() to remove all Contacts.
+ Added getLocalAddress() to return the IP address used by the Dispatcher.
+ Added startListening(), stopListening(), getListeningAddress(), and getListeningPort() methods. These methods will start and stop OSC message listening and return socket values for the listener as needed by the new NDEF system.
+ Added requestConnection() and requestMessages() to send out NDEF connection and message requests.
+ Added acceptMessage() (from the OSCListener interface) to accept various NDEF-related OSC messages.
+ Added getRelays() to return the Relay objects created by NDEF message exchanges.
+ Added getRejectedMessages() and clearRejectedMessages() to get message rejected NDEF message strings and to clear those message strings.

Junction

+ The setTarget() method now has a sibling that takes a Relay as an argument.
+ Added allowScalingWidth() and allowScalingHeight() to give finer control over scaling.
+ New getShape() method returns RECT or ELLIPSE in Processing.

+ Added changeWidth() and changeHeight() to give finer control over scaling.
+ Added limitScalingWidth() and limitScalingHeight().
+ Added getMinScalingWidth(), getMaxScalingWidth(), getMinScalingHeight(), and getMaxScalingHeight().
+ Added getJunctionCount() to return the number of subjunctions.
+ Added the clearContacts() method to remove all Contacts.
+ Added unmapMessage() to remove all mappings for a given message.

Relay

+ Added two varieties of addLong() to add long values to all messages or to a specific message.
+ Added containsMessage() to determine whether a message has been added.
+ Added replaceMessage() for swapping message Strings.

Action

+ Added the CONTACT_X, CONTACT_Y, CONTACT_XY, and CONTACT_R actions that allow for the mapping of Contact movements within the area of a Junction.

API Changes

Distpatcher

= Changed getJunctionArray() to getJunctions() for simplicity.

Junction

= The boolean fields rotatable, scalable, translatable, translateX, and translateY are no longer public. They have been replaced by the methods

allowRotation(), allowScaling(), allowTranslation(), allowTranslationX(), and allowTranslationY(). The new methods allow rotation, scaling, and translation setting to be inherited by subjunctions.

- The public booleans rotateClockwise and rotateCounterclockwise have been removed. Rotation can be limited by setting minimum and maximum values.

= The changeScale() method now takes two values, one for width and one for height. Scaling can still be done proportionally if the two values are the same.

= The getWidth() and getHeight() methods return the actual values for width and height. Previously, they returned those values multiplied by a scale factor. The scale factor has gone away.

= Changed limitTranslateX() and limitTranslateY() to limitTranslationX() and limitTranslationY(). This naming is more consistent.

= Changed getRotations() to getRotationCount() since this is a more accurate name.

= Changed getMinTranslateX(), getMaxTranslateX(), getMinTranslateY(), and getMaxTranslateY() to getMinTranslationX(), getMaxTranslationX(), getMinTranslationY(), and getMaxTranslationY().

= The updateJunction() method now takes two values for scaling, seperate floats for width and height.

= Changed getJunctionArray() to getJunctions().

= The addContact() method now takes a long value for a Contact id and the x and y values for the center of the Contact. The Contact is then created by addContact() rather being passed to it.

- The updateContact() method now only takes a long for the Contact id and the x and y values for the center. The values for speed and acceleration may return again.

= The removeContact() method now takes a primitive long value rather than the Long class.

= Changed getContactArray() to getContacts().

= Changed addMessage() to mapMessage() since this is a far more accurate name for this method.

- Removed the isMoving() method since it never really worked well.

- Removed the angle() method since it is no longer needed for determining rotation angles.

Relay

= Changed clearMessage() to resetMessage() since clear now means remove in this API.

= Changed removeAllMessages() to clearMessages() for API consistancy.

Contact

- The Contact() constructor now only takes the x and y values for the center.

- Removed setVelocityX(), setVelocityY(), setAcceleration(), getVelocityX(), getVelocityY() and getAcceleration(). They may return again some day.

- Removed the moving() method. It is not likely to return.

Action

= Changed ACTIVE to ACTIVATE since all actions should now be verbs.

Bug Fixes

= In limitTranslateY(), the useTranslateYLimit boolean was not being set to true. Instead, it was being set in limitTranslateX(). It is now set in the right method.

Internal Changes

Many small changes appear in this version. I'm just going to describe some of the more significant changes in Junctions.

Junction

= Inside of updateContact(), the Contact to be updated was removed from the map inside of Junction and then put back. This is not required and so the Contact simply receives new values from updateContact() without the extra steps.

= The scaling of Junctions now works differently internally. It still requires a two-Contact gesture externally. Now, the distance between the two Contacts is used to determine the scale changes. Before, the distance of the two contacts from the center was used. Which is silly and caused strange behaviors. The two Contacts have to be moving in order to begin scaling.

= Any field that is changed via a method is not private. Basically, all fields are now private.

+ Minimum and maximum values for width and height are not being set at 1 for minimums and bow width and bow height for maximums. The useTranslate_Limit booleans have been removed.

- The useTranslate_Limit methods have been removed as per the previous change.

= In setWidth() and setHeight(), the maximum and minimum values for are checked to insure that scaling is proportional.

= In addJunction(), limits are now inherited from the parent Junction.

+ All rotation, scaling and translation settings are now inherited in addJunction(). The settings can still be overridden by subjunctions.

= For rotation, in updateContact(), angles are determined with atan2. This is a much better way to handle one-Contact rotation than the law of cosines.

Android

The Android version of JunctionBox is part of this release. There are a number of internal changes to the Dispatcher, Junction and Relay classes. Here are the significant changes in terms of the API:

+ The Dispatcher now has the handleMotionEvent(MotionEvent event) method. This is used to get touch data in Android. Processing sketches MUST now call dispatchTouchEvent(MotionEvent ev) in the sketch with a call to handleMotionEvent() inside it. For example:
  public boolean dispatchTouchEvent(MotionEvent ev) dispatcher.handleMotionEvent(
  return true;
  If this call is not in the sketch, touch tracking will simply not work.

+ To import JunctionBox classes into a sketch use:
  import junctionbox.android.*;
  Note that this may change in a future release. The current namespace sep-

aration is, however, friendly for the Processing way of handling libraries.

+ All JunctionBox-based sketches MUST have the INTERNET permission set in either the Processing IDE or in the AndroidManifest.xml file.

  Other changes *should* be internal and therefore not problematic for porting sketches to Android. But it might be a good idea to check out the changes in Processing for Android:

  `http://wiki.processing.org/w/Android`

JunctionBox 0.9 - 15 May 2011

New Features

+ Added Junction.setLive() and Junction.isLive() to set whether a Junction will receive Contact from the Dispatcher. This overrides the previous feature of a public boolean called live since that boolean is now private. Having the set method allows subjunctions to inherit the live status.
+ Can now use new setToggle() to manually set the state of the toggle boolean in Junctions.

API Changes

= The Dispatcher's getJunctionArray() method now returns Junctions in the opposite order. This was done to be more consistant with Processing where graphical elements overlap with the most recently created element on top of the previous. So the most recently created Junction will now get Contact when it overlaps a previously created Junction in the same area.

= The cursor methods addTuioCursor(), updateTuioCursor() and removeTuioCursor() now check a array created from the Dispatcher's Junction list in reverse order. See the previous item for the reasoning behind this change.

- Removed Junction count-checking code from refresh() method.

+ Added changeCenterX(), changeCenterY() and setCenter() to Junctions. These new methods take delta values rather than set values for changing the center location.

+ Added changeAngle() to Junction for providing delta values for rotation angle.

= Changed toggle public field to setToggle() and getToggle() methods.

+ Added limitRotation() method for setting minimum and maximum angles.

+ New getJunctionArray() method returns any subjunctions of a Junction.

= In Junction, changed active() and moving() methods to isActive() and isMoving() for better naming.

- Removed the checkCounts() method from Junction since that functionality is handled elsewhere.

+ Added relayContactCount() and relayRotationCount() methods to Junction to replace checkCounts().

+ Added new SCALE_WIDTH and SCALE_HEIGHT actions to Junction.

- Removed, from Relay, all versions of setMesage(), setRange() and dump().

+ Added getIPAddress() and getPort() to Relay.

+ Added addMessage(), getMessages(), clearMessage(), removeMessage() and removeAllMessages() to Relay.

+ New methods for setting values for messages in Relay include addInte-

ger(), addFloat(), addString() and addBlob(). Each of these methods has variations for adding values to specific message or all messages. A version of addFloat() also has mapping parameters.

= In Relay, the send() method now has three variations: one to send a specified message, another to send an array of messages and the third to send all messages.

+ Added new SCALE_WIDTH and SCALE_HEIGHT values to Action.

Internal Changes

= Moved duplicated code from the various Dispatcher constructors into an initialize() method that created the Junction list and creates and connects the TuioClient.

= Changed the way in which Junctions are added to the list in the Dispatcher. Junctions are now added to the end of the list rather than the beginning.

= In the Dispatcher, addTuioCursor(), updateTuioCursor() and removeTuioCursor() all created their own array from the Junction list in order to avoid concurrency problems.

- Removed HashMap from Dispatcher that contained Contacts that were not added, updated or removed from Junctions. All Contact handling code from addTuioCursor, updateTuioCursor and removeTuioCursor has been removed. This functionality does not belong in the Dispatcher.

= Changed Junction's list of subjunctions from Vector to CopyOnWriteArrayList to void potential concurrency issues.

= In Junction, changed the on boolean to toggleOn to give it a more specific

and understandable name.

+ Added minAngle and maxAngle floats as well as a limitAngle boolean to determine whether to use the minAngle and maxAngle.

= Junctions now have a single Relay for sending messages. Where previously each actions had a list of Relays, now each actions has a list of OSC message Strings. These message Strings are then sent out via the Relay. Each list is a Vector of Strings.

+ Junctions now longer have translations limits set to the box width and height by default. Limits can still be set manually.

+ Since translation limits are no longer required, added useTranslateXLimit and useTranslateYLimit booleans to test whether limits have been set.

= Fixed setCenterX(), setCenterY(), setWidth() and setHeight() in Junction so that they obey their relevant minimum and maximum values.

= In Junctions, addContact() and removeContact() now call relayContactCount() if the number of Contacts has changed.

= Moved Junction code in updateCursor() to the various new changeX() methods.

= Added scaleWidthList and scaleHeightList as well as relayScaleWidth and relayScaleHeight to Junction to handle the new actions.

- From Relay, removed useRange boolean and argFloat.

- In Relay, added a port integer.

+ Added, to Relay, a ConcurrentHashMap called messageMap for holding multiple messages in a single Relay.

= Used InetAddres.getByName() to parse IP address in Relay constructor to avoid having to break up address with custom code.

JunctionBox 0.8 - 18 August 2010

Bug Fixes

+ In Junction.addContact(), added Contact to map after checking the map to determine whether or not the Contact is completely new. The ACTIVE relay should now work properly.
= Fixed Junction.inside() test for clockwise rotation of rectangles and for ellipses both rotated and non-rotated.
= The Simulator now sends better values for velocity and acceleration of Contacts.

New Features

+ Added getContactArray() method to Junction that returns an array of copies of the current Contacts for that Junction. The values of Contact.getX() and Contact.getY() are scaled by the boxWidth and boxHeight respectively. This means that the scaling will not have to be done in Processing.
+ Added setAngle() to Junction to allow for directly setting the angle.
+ The Contact class now has the setVelocityX(), setVelocityY(), setAcceleration(), getVelocityX(), getVelocityY(), getAcceleration() and moving() methods. These values are obtained from the TuioContainer class.
+ Added new Action.TRANSLATE_XY. This action will send messages with both x and y values in the same message.

+ A Junction can now handle relaying TRANSLATE_XY values via the new Action described above.
+ Added new Relay.send() methods that take no argument, two floats and an Object[].
+ Junction.updateJunction() will now update any subjunctions with values for center location (translation), rotation or scaling. This allows subjuctions to inherit behavior from parent Junctions. This will happen automatically when subjuctions are added using Junction.addJunction(). Note that this method is protected.

Internal Changes

= Changed contactMap from Hashtable to ConcurrentHashMap in order to allow for Contact lists to be created from contactMap without having to worry about the map being updated while the list is being created.
= Several methods were changed from public to protected including Junction.addContact(), Junction.updateContact(), Junction.removeContact(), Junction.containsContact(), Junction.getContact(), Junction.inside(), Contact.setX() and Contact.setY().

JunctionBox 0.7 - 8 June 2010

New Features

+ Added new actions CONTACT_COUNT and ROTATION_COUNT. See next item for details.
+ Added checkCounts() to Junction that enables the Contact count to be relayed via CONTACT_COUNT and the current rotation count to be re-

layed via ROTATION_COUNT. The new checkCounts() method is called from Dispatcher.refresh(), a previously unused method required by TuioListener. By using refresh(), count checks can be made more regularly.

+ Added createJunction() to the Dispatcher. This allows for Junction creation and adding in a single line. Junctions can still be created using the new constructor and added to the Dispatcher as before.

+ The Dispatcher now has multiple constructors. The new constructors allow for various combinations of new parameters, including box width/height and target address/port. If parameters are provided via the new constructors, they will be passed to Junctions.

+ Added setTarget() to Junction that sets a destination target for messages sent from the Junction.

+ Added addMessage() to Junction that allows for mapping actions to messages directly, having the Junction create the Relay automatically. This can save lines of code in Processing situations in which only one target is needed for a given Junction.

API Changes

- Removed the Constants class since it was no longer useful. See next item.

= Removed the Action enum from the Contstants class. This enum still carries the same actions as before but is now called as Action.ROTATE instead of Constants.Action.ROTATE.

+ Junctions now implement PConstants. Now shapes can be designated via Processing constants RECT and ELLIPSE instead of Constants.Shape.RECT and Constants.Shape.ELLIPSE.

+ Simulator now implements PConstants for greater Processing integration. Simulator can now check for LEFT, RIGHT or CENTER in the mouse method as in Processing.

= Changed Relay.setAddress() to Relay.setMessage(). This does not strictly follow the OSC spec but it makes more sense in the larger context of JunctionBox.

Internal Changes

- Removed excessive use of the "this" keyword, especially from Junction. This will hopefully make the code a little easier to read.

= Changed parameter target to address. A target should be an address and a port.

= Changed argument names in Simulator.mouse() to pressed, button, x, y, px, py to avoid confusion with the equivalent values in Processing and their use within the Simulator.

JunctionBox 0.6 - 13 May 2010

Bug Fixes

+ In Junction, fixed containsContact() so that it takes into account sub-junctions.

New Features

+ Added a Simulator class that can simulate TUIO message directly in Processing.
+ Added dump() method to Relay that allow current OSC arguments to be output as Strings.
+ Added moving() method to Junction that returns true if the Junction is moving. The output of moving() is not terribly accurate, so caution should be used in relying on it.
+ Added a Contact container to the Dispatcher so that any Contacts that are not associated with a Junction can still be tracked. Nothing is being done with the extra Contacts but this may change in future versions.
+ Added a live boolean field that can deactivate a Junction, preventing it from picking up any new Contacts.
+ Added the orderJunction() to the Dispatcher for setting the order of Junctions in the Dispatcher's main list. This can be important since a Contact can be assigned to only one Junction and that assignment happens in order based on the Dispatcher's list.
+ Added getJunctionArray() to the Dispatcher for getting Junctions in the order of the Dispatcher's main list. This can be used with orderJunction() to change the output order of Junctions in Processing.

API Changes

= Changed handling of OSC messages in Relays. The send() method no longer accepts any OSC type. Only float, int and String values can be sent using overloaded versions of send().
= setTranslateX() and setTranslateY() are now called limitTranslateX() and

limitTranslateY() to better reflect their purpose.

Internal Changes

= In Dispatcher, changed list of Junctions from Vector to CopyOnWriteArrayList. This prevents locking of list but allows for concurrent access without throwing an exception.

JunctionBox 0.5 - 2 April 2010

Initial Release