FlexiGesture: A sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control

by

David Jeffrey Merrill

B.S. Symbolic Systems, Stanford University (2000)
M.S. Computer Science, Stanford University (2002)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNLOGY

June 2004

Author _____
Program in Media Arts and Sciences
May 14, 2004

Certified by _____
Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Sony Career Development Professor
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by _____
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

FlexiGesture: A sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control

by

David Jeffrey Merrill

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on May 14, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

Acoustic musical instruments have traditionally featured static mappings from input gesture to output sound, their input affordances being tied to the physics of their sound-production mechanism. More recently, the advent of digital sound synthesizers and electronic music controllers has abolished the tight coupling between input gesture and resultant sound, making an exponentially large range of input-to-output mappings possible, as well as an infinite set of possible timbres. This revolutionary change in the way sound can be produced and controlled brings with it the burden of design: Compelling and natural mappings from gesture to sound now must be created in order to create a playable electronic music instrument. The goal of this thesis is to present a device that allows flexible assignment of input gesture to output sound, so acting as a laboratory to help further understanding about the connection from gesture to sound.

An embodied multi-degree-of-freedom gestural input device was constructed. The device was built to support six-degree-of-freedom inertial sensing, five isometric buttons, two digital buttons, two-axis bend sensing, isometric rotation sensing, and isotonic electric field sensing of position. Software was written to handle the incoming serial data, and to implement a trainable interface by which a user can explore the sounds possible with the device, associate a custom inertial gesture with a sound for later playback, make custom input degree-of-freedom (DOF) to effect modulation mappings, and play with the resulting configuration.

A user study with 25 subjects was run to evaluate the system in terms of its engaging-ness, enjoyability, ability to inspire interest in future play and performance, ease of gesturing and novelty. In addition to these subjective measures, implicit data was collected about the types of gesture-to-sound and input-DOF-to-effect mappings that the subjects created. Favorable and interesting results were found in the data from the study, indicating that a flexible trainable musical instrument is not only a compelling performance tool, but is a useful laboratory for understanding the connection between human gesture and sound.

Thesis Supervisor: Joseph A. Paradiso
Title: Associate Professor of Media Arts and Sciences

FlexiGesture: A sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control

by

David Jeffrey Merrill

The following people served as readers for this thesis:

Thesis Reader _____

Bruce M. Blumberg
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Massachusetts Institute of Technology

Thesis Reader _____

Perry R. Cook
Associate Professor of Computer Science and Music
Princeton University

Acknowledgements

I'd like to give acknowledgement to the following people, who have each been an important part of my progress on this project.

To **Joe Paradiso**, for believing in my ability to make this research project happen, and for his continuing support and advice along the way. I have gained much insight into electronics and music through my work with Joe, and for that experience I am very grateful.

To **Bruce Blumberg** and **Perry Cook**, my gracious readers, for providing early-stage ideas and enthusiasm for this project, as well as later-stage advice.

To **Stacy Morris**, who helped me get up and running with the stack hardware, and gave me bits of her time when there weren't really any spare bits.

To the staff at the Media Lab, especially **Linda Peterson** and **Lisa Lieberson**, for their invaluable administrative help and counsel during the past year and a half.

To **Yuri Ivanov**, mentor and friend, for his generous advice and interest in this project.

To **Ted Selker**, for kicking off this adventure by bringing me to the Media Lab in the first place, and for teaching me a great deal about building and inventing along the way.

To my colleagues in the **Responsive Environments Group**, who have been good friends and wise sources of "how-to" expertise.

To my **parents**, who always expected me to achieve and gave me the love and support to do so, to my **sister** whose talent for "real music" is unmatched, and to my **brother** whose creative pursuits are off to a great start. I appreciate all of you more than you'll ever know.

Finally, to **Amy**, whom I adore and whose companionship is selfless and true.

Thanks everyone!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction: On old and new musical instruments

The history of musical instruments has long been about the sounds they make, the materials they are made from, and the affordances that they provide the performer. From Chinese bone flutes dated at 9000 years old [Zhang, et al. 1999], to Italian violins from the 16th century [NMM website], the art of making an expressive instrument has consistently taken advantage of the current state-of-the-art in materials and technique. Up until the past 100 years, the materials available for building musical instruments have been passives like wood and bone, stone and metal. Instrument makers were craftspeople that had mastered the art of sculpting and joining these materials, and their instruments featured affordances like strings to bow, reeds to blow, and keys to strike.

The emergence of electronics and computers, the recent shrinking in size of microcontrollers and electronic components, and the development of the fields of artificial intelligence and pattern recognition are allowing the materials and technique of instrument building to diverge from tradition in a profound and discontinuous manner. Handheld instruments can now be active, having many different types of built-in sensing, processing and communication abilities. In addition, the generation of sound is no longer tied to the physics of a musical instrument. Whereas a 20th century horn produced sound by exciting a resonant mode of oscillation in an air chamber made of brass, an electronic musical instrument today can synthesize sound algorithmically, generating waveforms with rules that are not governed by any physical constraint.

The opportunity created by these new instrument-making capabilities brings with it a corresponding degree of complexity. The opportunity comes from the fact that the sound produced by the device is no longer coupled to its physical design. Small, cheap sensors can be densely packed into an interface, creating a multi-degree-of-freedom device with input affordances that have no inherent mapping to sound. This decoupling makes the designer's imagination the only limit to how a performer can play a device and what sounds can be produced. The new complexity is that with the exploded range of choices faced by an instrument-builder, it is not obvious how to decide the ways that the instrument will sense the performer's gesture and how that manipulation will be turned into sound.

This thesis proposes to solve this problem by introducing a new sensor-rich electronic music controller device, and a novel gesture-to-sound mapping scheme that allows the device to adapt to the gestures of the user. I hope that this system will serve as a model for current and future input device designers – musical and otherwise – in building a personalizable and expressive physical input device.

1.1 Acoustic Instruments

An acoustic musical instrument is a device that is played through physical manipulation by a person, and that creates sound through this manipulation solely by virtue of its physical construction. Various taxonomies have been proposed by scholars over the years for acoustic musical instruments, and these categorizations are typically based on the mechanism by which the sound is produced. Mahillion grouped acoustic instruments into autophones (bells, xylophones, music boxes), aerophones (wind), chordophones (stringed), and membranophones (drums) [Mahillion 1893]. Sachs refined upon this categorization later, changing the name "autophone" to "idiophone" to avoid making the suggestion that this class of devices would play themselves [Sachs 1913]. A recent taxonomy by Kvifte [Kvifte 1989] is based more upon the physical manipulations that a performer uses to actuate the instrument, such as plucking, bowing, or striking[1].

Building and playing musical instruments have long been a part of the human experience. The earliest musical instruments that we have evidence of are playable flutes discovered in China that have been dated at 9000 years old [Zhang, et al. 1999]. To put this figure in perspective, the earliest known wheel, made of stone and uncovered in modern-day Iraq, has been dated at around 5500 years old. This long history of musical instrument design speaks to the centrality of music and musical-instrument construction in the human experience. As a result of these many years of human craftsmanship directed towards building playable instruments, *acoustic* musical instruments have converged towards a number of mature designs today. Modern versions of acoustic instruments like the violin, flute, and piano have the benefit of hundreds or even thousands of years of iterations informing their current design. Examination of surviving acoustic instruments from just the past few centuries (guitars, for instance) shows that constant changes and refinements to certain instruments have been made, while others that matured early (violin, for instance) have remained relatively unchanged for hundreds of years.

[1] In this case, plucked and bowed violin are in separate families [Goudeseune 2001]

1.1.1 The coupling of the physical body to sound in acoustic instruments

The sounds produced by traditional acoustic instruments are intimately connected to the component materials, the way that these materials are put together, and the way that a performer interacts with the device. We can examine this claim by taking a guitar as an example of a stringed acoustic instrument. The body of a guitar is typically made of a wood such as mahogany, spruce, maple, cedar, or some variety of rosewood [Santa Cruz Guitar Company website], and the strings are made of steel or nylon. In order to create sound with a guitar, a performer can strum, tap, pluck, or scrape a pick across the strings. The performer can also tap on the body in a percussive way. These manipulations are just the most common subset of the enormous range of manipulations that performers over the years have used to create sounds with guitars [Frith] [Bailey]. The important point about the example is that all of the sounds that are possible to produce with an acoustic guitar are linked to the physics of the embodied instrument. These sounds are produced when modes of vibration in the resonant cavity of the body become excited by manipulations made to the strings or to the body directly. These vibrations project sound out into the open air.

Performers have always found new and unusual ways to manipulate acoustic instruments in order to create novel sounds with them (see the discussion of affordances below), but ultimately the sounds that are possible are constrained by the physics of the instrument. When exciting a given physical object to produce audible vibrations, the physical structure of the object necessarily constrains the space of possible sounds.

1.2 Multi-DOF human-computer interfaces, and the decoupling of input and output

The myriad ways that we interact with musical instruments has changed dramatically over the past century. The following section will explore the idea of input affordances, degrees-of-freedom, and modern synthesizers. It will provide the backdrop against which the mapping problem is framed in the next section.

1.2.1 Affordances

Don Norman defined affordances as "the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used." [Norman 1988] The affordances of acoustic musical instruments are typically tied to the sound-production mechanism. For instance, the strings of a guitar afford plucking, fretting, and strumming, while the mouthpiece of a saxophone affords blowing, and its keys afford pressing. Each of these ways to use the instrument has to do directly with the physics of the sound-production mechanism. A sophisticated player of an acoustic instrument is aware of many more subtle affordances of their instrument, learned and discovered through years of experimentation and familiarity with the device. For instance, the body of a stand-up bass can be tapped on in a percussive manner to create a rhythm, and the neck of a guitar can be bent to slightly detune the strings. These subtle affordances may not have been explicitly designed into the device, but rather they "fall out" of the way it is built. Nonetheless, the subtle affordances contribute to the complexity of the device, and the resulting expressiveness that is achievable when playing.

Electronic music instruments, in contrast to their acoustic predecessors, tend to have fewer subtle affordances. They are typically designed to measure a performer's gesture with sensors or switches, converting the instantaneous analog state of the device into a discrete symbolic representation. This symbolic representation is then transmitted to a sound synthesizer, which generates an output waveform in response. A crucial bottleneck in this transmission of information from input gesture to output sound is the intermediate symbolic representation. Whereas an acoustic instrument has the benefit of the rich set of affordances that fall out of the direct and physically based connection between gesture and sound, the input affordances of an electronic music instrument typically have to be designed into the system explicitly. Unusual ways of playing electronic music controllers are inevitably discovered by creative performers, but since every sensor or switch must be placed intentionally by the instrument-designer, there are necessarily fewer of these "happy accident" style affordances than the number that fall out of the design of an acoustic instrument.

One way that performers with electronic music controllers overcome a lack of affordances is to develop advanced playing technique. The theremin – the first electronic musical instrument to gain any significant degree of popularity – only really had 2 affordances, the performer's body distance from two receivers on the instrument was linked to volume and pitch [Galeyev 1995]. However, theremin virtuosos over the years have developed amazingly fine hand-and-upper-body motor control that allows for a high degree of expression and precision. A possible way for an electronic music instrument to mitigate its lack of accidental affordances is for its designer to build in a large number of explicitly placed affordances. The Sensor Shoe [Paradiso et al. 2000] was a wearable shoe controller with 16+ sensors that were mapped to musical events in a number of separate performances. Electronic sensors and components are becoming smaller and smaller, and thus today it becomes possible to greatly increase the number of sensing elements contained by a single electronic music instrument. This is the strategy employed by the author, and the result is a multi-degree-of-freedom device.

1.2.2 Multi-degree-of-freedom interfaces

A multi-degree-of-freedom (multi-DOF) interface is one that presents a number of different input affordances to the user. These affordances are typically linked to an input degree-of-freedom present in the instrument. The distinction between an affordance and an input degree-of-freedom is that an input degree-of-freedom is a manipulation of the device that is *measured* by a switch or continuous sensor, while an affordance can be any aspect of the device that affects how it can be used. For example, an input degree of freedom of a car steering wheel is the amount of rotation applied to the wheel, while an affordance might be the horizontal segments of the wheel that allow the driver's hands to be rested on them during calm driving. Turning and resting are both ways that a driver interacts with the system, but while the degree of rotation is sensed and used by the system, the hand-resting usage is not. Multi-degree-of-freedom computer interfaces offer exciting possibilities for electronic musical instrument and interface designers, because

number and diversity of sensors available today creates the opportunity to build a device that can be as expressive – albeit with different affordances – as its acoustic counterparts.

1.2.3 Synthesizers: finally, arbitrary output

The emergence of electronic and digital audio synthesizers in the last century is enabling radical and discontinuous change in the way that musical sounds are made and controlled. Analog synthesizers use hardware oscillators, filters, analog signal conditioning and control electronics to synthesize dynamic spectra-rich tones, while digital synthesizers use algorithmic rules and digitized representations of a sound to create their output. The variety of sound synthesis techniques in use today is beyond the scope of this work, but most notable is the fact that these techniques allow arbitrary waveforms to be produced and turned into audible sound. No longer is the sound generated by a system dependent on the physics of a physical object; rather, the possible sounds are subject only to the limits of the skill and imagination of the audio designer.

1.2.4 Decoupled at last: implementations and implications

Cheap and miniaturized electronic sensors and components, coupled with modern sound synthesizers offer a discontinuous leap in musical instrument design because they break down the traditionally mandatory coupling between the way a performer plays an instrument (affordances and input degrees-of-freedom) and the way it produces audible sound (synthesis). In order to create an electronic music controller though, there must necessarily be some connection between the performer's input gesture and the system's output sound. Early electronic music controllers like the theremin featured a direct mapping from input to output, the sensed level of some input degree of freedom tied directly to an obvious parameter like pitch of volume. With the advent of computers, the link between input and output can be made more abstract and "stateful", with sophisticated symbolic processing happening between the input sensor and the synthesizer. Over the years there have been several such "glue" implementations, and in the following paragraphs we will look specifically at two of them, MIDI and OSC.

MIDI is a hardware specification and messaging protocol that was designed in the early 1980's to allow arbitrary controller/synthesizer pairings. MIDI's messaging protocol is digital, and messages are typically 2-3 bytes in length. These messages are transmitted via an asynchronous serial interface that runs at 31,250 bits/sec (baud). MIDI has immense momentum today, being implemented in thousands of commercial hardware and software products, and it is certainly the most popular example of the decoupling of input control and sound synthesis in electronic music controllers. MIDI revolutionized the music industry because it allows physically different controllers to all "speak the same language", providing an opportunity for a new "mix-and-match" modularity in coupling controllers to synthesizers. The price of this modularity though, is a number of limitations that the protocol imposes on latency and precision for continuous control. For instance, the "data byte" of a standard MIDI message only has 7 bits to work with, since the state of the MSB determines the difference between a status and data byte. This means that at most 128 unique values can be transmitted for a particular control. Another limitation is that on a single MIDI circuit there can be at most 16 devices daisy-chained together, and the latency through this type of loop can reach perceptually unacceptable levels.

Awareness of the control bottleneck-related deficiencies of a MIDI has inspired work in various directions. One approach is to "re-couple" the controller's affordances to the sound production mechanism by co-designing the controller and the sound synthesis algorithm together [Cook 2003]. A more basic solution is implemented by the OpenSound Control (OSC) protocol, which is "a protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology" [OpenSound Control webpage]. OSC is more flexible than MIDI in terms of how much data can be packed into messages, how messages are addressed to recipients (includes string-matching style wildcards), and is hardware-layer agnostic, allowing OSC packets to be routed over a gigabit Ethernet system just as easily as a RS232 serial link. Like MIDI, OSC is an open and royalty-free specification, and its features make OSC a natural successor to MIDI.

Regardless of the intermediate messaging protocol used, the decoupling of the performer's sensed gesture from the commands sent to the synthesizer gives electronic music controllers modes of interaction that are not constrained by the instrument's physical acoustics. This decoupling of input and output featured by electronic music controllers and synthesizers is shared by any modern human-computer interface.

1.3 The modern mapping problem, and related work

For electronic musical instruments, the modern decoupling of gestural input and synthesized output has made possible convergent (many-to-one) or divergent (one-to-many) mappings of controller degrees of freedom onto synthesizer parameters [Rovan et al. 1997]. With N input degrees of freedom, and M synthesizer parameters, there are N*M possible direct mappings – not to mention the infinite number of more "stateful" mappings. Many people find the built-in mappings of traditional acoustic instruments difficult enough to master; with an exponentially expanded and potentially arbitrary "mapping space" comes the possibility of even lower usability.

The most transparent approach to mapping the outputs from a music controller to the inputs of a synthesizer is to create behavior that mimics that of a similar acoustic instrument. Bernd Schoner built a system that learned the low-level associations between the sensed input state of a violin and the acoustic output in low-level, data-driven manner. Once Schoner's system had enough examples to characterize the connection between input gesture and output sound, it was capable of replicating the behavior of a violin [Schoner 1999]. Schoner's approach has similarities to the current work, in the sense that his system learns to associate appropriate output with the sensed gesture of a performer.

Looking further than mappings that mimic acoustic instrument behavior, the growth in the possibility space for connecting gesture to sound brings with it the opportunity for innovative designs that can make a computer-assisted activity like music-creation more enjoyable, efficient, and expressive than ever before.

Explicit mapping strategies for electronic music controllers have achieved some success within the enthusiast community, but thus far no truly new music controller has gained broad acceptance. New electronic instruments that *have* entered the mainstream, like the electronic keyboard and electric guitar, tend to borrow their mappings from existing acoustic instruments. In addition, synthesizer manufacturers that get returned units for servicing find that users very seldom alter the preset mappings that ship with a device. Does this mean that synthesizer manufacturers ship their products with presets so brilliant that users never desire to customize? Perhaps more likely is the hypothesis that users find it too difficult to create their own mappings, and thus end up leaving the presets unchanged.

Research on mapping [Hunt, Wanderley & Kirk 2000] [Hunt & Kirk 2000] has indicated that for some multiparametric control tasks people prefer fewer, more abstract degrees of freedom to many, simple (separated) degrees of freedom. Hunt, Wanderley and Kirk recommend complex mappings such as cross-coupling of input parameters to synthesis parameters, and the use of derivatives of input parameters related to the performer's energy. In addition, it has been shown that in certain cases nonlinear couplings of input parameters can be effective [Rovan et al. 1997]. It has been suggested that the mapping scheme is perhaps the *most* important element to a controller/synthesizer system [Hunt, Wanderley and Paradis 2002]. Another direction is a geometric mapping in euclidian spaces of a small number of input parameters onto a larger number of synthesizer parameters [Garnett & Goudeseune 1999], with a method for automatically generating perceptual parameters for the resulting points in output space called the *timbre rover*. A related implementation defines a rough mapping from a finite number of input space configurations to output space configurations, and a corresponding interpolation scheme for generating mappings for in-between values [Bowler and Purvis 1990].

1.4 Adaptive user interfaces

Adaptive user interfaces are a branch of the field of Human-Computer Interaction that borrow techniques from Artificial Intelligence. The goal of an adaptive user interface is to "personalize interfaces, based on observation of user activity." [Langley 1997] Currently most adaptive user interfaces operate on non-expressive domains like scheduling, information access, and commercial applications [AAAI 2000 Spring Symposium], but there has also been work in gestural understanding [Ivanov 2002] [Pavlović and Rehg 2000] [Wilson 2000] and mapping. Robust handwriting recognition has been a longstanding challenge in the artificial intelligence/pattern recognition community [Bledsoe and Browning 1959], and hand-gesture recognition with neural networks has been used to drive a speech synthesizer in real-time [Fels & Hinton 1995]. Blumberg has created a behavioral learning system called Alpha Wolf in which onscreen animal characters learn novel behavior patterns by example, as a human "trainer" leads them through novel actions [Blumberg 2002]. The following section will review work in adaptive interfaces specifically from the musical domain.

1.5 Existing algorithmic and adaptive musical interfaces

The idea of creating a musical interface that learns, evolves, or adapts to the user is not entirely new. There are a number of systems that have been built that explore ways in which machine-learning, genetic algorithms and pattern-recognition algorithms could be applied to the creation of music. The following section will survey some of these systems.

1.5.1 Style Imitation

> *The ability to construct a musical theory from examples presents a great intellectual challenge that, if successfully met, could foster a range of new creative applications*
>
> -Dubnov, Assayag, Lartillot and Bejerano, IEEE 2003

Imitating the musical style of a particular composer or performer is the focus of several musical learning systems. The way these systems typically operate is to read in a corpus of musical examples – commonly in MIDI file format and consisting of a sequence of notes scheduled in time - and generate a statistical representation of the corpus. Once the representation has been made, the generative process uses it to produce a new sequence of notes that has similar statistical characteristics as the original corpus, but that is not a note-for-note reproduction of any of the original inputs. Dubnov and Assayag's system models a style as a lexicon of motifs and their associated prediction probabilities. Generating new content consists of "stochastically browsing" the prediction tree to decide on the next phrase at a given step [Dubnov et al. 2003]. David Cope's work deconstructs pieces from the corpus to identify signatures, melodic snippets that are characteristic of the corpus [Cope 1990]. Output generation in Cope's system is a process of recombination, with the reuse of these signatures adding a degree of naturalness to the music. Pachet's Continuator system brings a real-time component to the style imitation, generating immediate, stylistically-related responses to musical phrases without the use of a large preprocessed corpus [Pachet 2002].

1.5.2 Adaptive signal processing

The phrase "adaptive signal processing" usually refers to a class of algorithms that adjust the parameters of a filter that is being applied to a stream of digital data. The adjustment is guided by a feedback loop in order to maximize a particular fitness function applied to the output [Widrow & Stearns 1985]. In the present context however, a more pertinent type of adaptive signal processing is embodied by Brad Garton's system Elthar [Garton 1989]. Elthar is an audio signal processing interface that allows for natural language input, and is designed to function with "incomplete or inexact specification of parameters." When given an underspecified directive, Elthar consults a probabilistic knowledge base that has been compiled based on a history of interaction with the current user to fill in the needed values. Elthar also contains an analogy mechanism that allows it to learn descriptive attributes.

1.5.3 Evolution-inspired systems

Genetic algorithms are finding a niche in the problem areas of state-space exploration and optimization [Koza et al. 1999]. Mandelis's Genophone software [Mandelis 2001] implements the artificial life paradigm by performing "selective breeding" on parameter populations in order to choose values for sound synthesis that are pleasing to a user. The important aspect of this work from the current point of view is that it allows the user to explore a large parameter space, but without requiring intimate knowledge of the sound synthesis algorithms. Mandelis's physical setup consists of a glove with bend-sensors on the fingers connected to a computer, and a Korg Prophesy synthesizer to create the sounds.

1.5.4 Other learning systems

Some attempts to do better than the manual specification of explicit mappings for electronic music controllers have included the use of neural networks to map hand gestures onto synthesizer parameters [Modler 2000], [Lee, Freed & Wessel 1991]. Cont, Coduys and Henry used Matlab to train neural networks (that ran in Pure Data [Pure Data website]) to recognize gestures captured by body-mounted sensors [Cont, et al. 2004]. Szilas and Cadoz proposed an analysis method for physical model networks that was based on connectionist learning algorithms [Szilas & Cadoz 1993]. Schoner's system for learning the mapping from physical gestures to a synthesized violin output was discussed earlier. Some commercial audio synthesizers feature a way to assign midi controllers to synthesis parameters. For instance, the Lexicon MPX110 [Lexicon MPX110 website] can be put into "Learn Mode" in which twiddling one of the knobs on the front panel and sending a MIDI controller message to the unit will create mapping between the particular MIDI controller and the knob's effect. The list continues, but an interesting pattern in the existing body of work is the minimal attention paid to the form of the physical device in these learning and classifying systems. Perhaps David Wessel said it best in the following excerpt from a 1991 issue of the Computer Music Journal [Wessel 1991].

Musicians often speak of a rather special and very personal relationship with their instrument. Indeed, many instrumentalists adapt the instrument physically to particularities of their playing style – choosing the bridge, string, bow, or the mouthpiece reed combinations, and so on. On more poetic occasions a musician will speak as if the instrument has come to know something of its player. It would seem quite natural then to think about intelligent instruments that could adapt in some automated way to a personal playing style.

-David Wessel

1.6 On embodied instruments

One of the early design decisions in formulating this research project was how much of an "object" the final product should be, and how different the device should be from existing interfaces. Many of the existing systems cited here are entirely software-based. Of the systems that do have a physical component, they tend to employ wearable sensors (data-gloves, bend-sensors, etc.) or use a traditional musical interface. Indeed, a look at the entire field of adaptive musical interfaces will produce surprisingly few new devices that have been built expressly to be entirely new "smart music controllers". The danger in co-opting existing interfaces for a new task is that a well-known controller produces a set of assumptions in both performer and audience about how it should be played, and about what kind of music will be possible with it. Joel Chadabe made the following comments about the piano-style keyboard:

Like every performance device, a keyboard's structure and, consequently, the physical gestures required to play it, cause us to think in a certain way about the music that we play. A keyboard is structurally a discontinuous controller which leads us to think of music as separate notes; and we play it with our fingers, which suggests a note-after-note data entry approach. Deriving from a long history of western music and western tuning systems, keyboards satisfy musical needs when those needs are consistent with with conventional musical values. When a musical concept is unconventional, however, the use of a particular performance device should be carefully considered.

- Joel Chadabe [Chadabe 2000]

Chadabe also emphasizes that a performance device must enable the audience to "perceive a meaningful relationship between the performer's gestures and the musical result, highlighting the extreme lack of this relationship in the performances of the growing number of "laptop musicians". Speaking to the specialness of embodiedness, Ishii has written about the importance of physical artifacts and the "vestiges of physical presence" that are left perceptible when one experiences a handmade piece of work. [Ishii 1998] In reflecting on his childhood visit to view the original manuscript of a poem by a famous Japanese author, Ishii claims that these vestiges become fewer the more digital and consequently "dry" an object or work becomes. Ishii also writes about the "co-evolution" of user and device in the context of ping-pong paddles that are customized by users explicitly (by scraping and sanding) and implicitly (by patterns of wear) over the years [Ishii, et al. 1999]. Bob Ostertag provides a nice connection between Ishii's points, and the design and construction of music controllers in his article "Human Bodies, Computer Music" [Ostertag 2002]. Ostertag quotes his friend Pierre Hébert, writing:

> *"the measure of a work of art is whether one can sense in it the presence of the artist's body. If so, then it is a success, and if not, it's a failure."*
>
> -Pierre Hébert, quoted by Bob Ostertag

Ostertag goes on to clarify that his own view isn't quite as extreme as Hébert's, but that the physicality of the performer/instrument relationship is an important component to any performance. In fact, many musicians talk of having a special connection to their instrument. BB King had endless praise for his favorite guitar (which he named "Lucille"), and Yo Yo Ma and "Petunia" (his Cello) are inseparable. Along the lines of Ishii's reflection on ping pong paddles acquire, musical instruments similarly come to show the personalized wear patterns of their owners, becoming more and more "personalized" over the years. This collection of ideas made it clear that constructing a new, embodied music controller would be preferable to building a software-only system, or attempting to adapt some existing interface to the design.

1.7 On human motion, emotion and musical expression

Manfred Clynes has spent his lifetime researching "the relationships between music and the brain" [Marrin 1996]. In the 1950's and 1960's, Clynes founded the field of Scentics which, in his own words, is "the study of genetically programmed dynamic forms of emotional expression" [Clynes 1977]. Building his own hardware system, the sentograph, Clynes measured gestural responses from the fingertip to emotional experiences like listening music. He ran studies on populations around the world, discovering universal patterns of response to musical forms. Clynes' work indicates a universality in human gestural responses to music, and is the inspiration for a part of the current work. In the current project we have looked for universal patterns in how people associate gesture and sound, both in terms of the shapes and energies of gestures that they use to trigger a certain sound, and in what types of manipulations they assign to control various sound-processing effects.

1.8 Project goals

The overall goal of this research project is to build an embodied electronic musical instrument that adapts to the player. The claim is made that the low cost and small size of electronic sensors and other components, coupled with the current processing power possessed by everyday personal computers makes the time ripe to explore ways that small, physical music controllers can learn personalized gestures and mappings from a user. Rather than the person having to learn the device, the device can begin to learn the person. As a step towards demonstrating this claim, a new electronic music controller and associated learning and interface system was designed and built. A study was conducted that collected users' subjective responses to the system as well as implicit data from their training and play with the system.

The embodied component of the system, the controller, is a multi-degree-of-freedom hand-held device that was designed from the ground up to have a number of contact and non-contact based, continuous affordances. It features three axes each of acceleration and

rotation sensing, a number of force-sensitive-resistor and potentiometer inputs, two back-to-back bend-sensor pairs, and an electric field sensing apparatus. The data from the sensors is collected by a microcontroller and transmitted serially to an external computer, where it is analyzed and mapped to auditory outputs. The analysis that happens on the remote PC includes dynamic time-warping for classification of gestures, windowed variance tracking for activity detection, and scaling of continuous-control data to match the range of control afforded by the synthesis algorithms.

The usage scenario is as follows: A user begins by exploring the sound-space that the system affords. The purpose of this exploration is to identify sounds that they would like to be able to trigger during a performance. In the present implementation the sounds available are a number of digitally sampled waveforms taken from drums and other percussive instruments, sleigh-bells, a turntable "scratch" sound, some stringed and wind instruments, and even white noise. The exploration is a linear traversal of the sounds, using a toggle button on the handle of the controller. When the user discovers a sound that they like, they train a gesture, or a number of gestures by squeezing on a "trigger" button while moving the device through a physical motion. The system builds a model of each gesture class, and keeps these models associated with appropriate sound. After making a number of these gesture-sound associations, the user can select audio-processing effects with which they would like to associate an input affordance. During each effect training interaction, the user hears one of the sounds from the gesture-training interaction, but with an oscillating amount of the effect applied to it, meaning that the sound might be soaked in an oscillating reverb, distortion or flange effect, etc.. These "variations" on the original sounds are invitations for the user to map a continuous input on the device to the associated effect. The mapping is accomplished by example, with the user exciting the degree of freedom that they want to associate with the effect. (see chapter 3 for specific details on this process) Once trained, the system can be used in a performance to trigger and modify sounds. Triggering a sound is accomplished by executing the associated gesture, and modification is accomplished by manipulating the associated continuous control.

This system is meant to accomplish two goals. First, it is intended to demonstrate a new way of mapping human gesture to sonic output on a novel, multi-degree of freedom electronic music controller. Second, it is meant more broadly to provide an example of an adaptive, embodied user interface that develops a personalized set of affordances for a user, with a minimal training phase. The author hopes that this example will be a point of reference for further work in this area.

Chapter 2 will discuss the design and construction of the physical controller, and will show a sample of the data stream generated by the device. Chapter 3 will describe the software, including the graphical user interface, the pattern recognition module, and the "glue code" that implements the interaction. Chapter 4 presents the design and results of the user study, and Chapter 5 concludes with lessons learned and ideas for future work.

Chapter 2

Hardware: Building the physical device

The physical device is an electro-mechanical sensing and data-transmission platform. It provides a tactile and free-gesture, multi-degree-of-freedom interface to the user and transmits real-time sensor data back to the host computer that is interpreted by software. The initial choice and development of the sensing platform will be discussed in some detail, including several modifications made to existing and emerging designs. The design of the physical device and its affordances will also be discussed.

2.1 Summary and goals

The design goal for the device was to implement an embodied, sensor-rich system that would be flexible enough to enable a wide range of input affordances and input-to-output mappings. The large number of sensors would provide a multi-degree-of-freedom interface, allowing the user to apply many different types of gesture in order to control sounds.

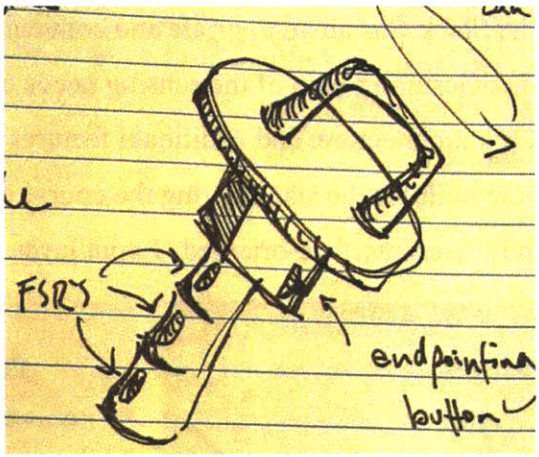

Figure 2-1: An early brainstorming sketch of the device

2.2 Sensors and Communication: the Stack platform

In order to support the high degree-of-freedom interaction, a sensor-rich platform was needed to capture and transmit the data to a PC for processing, gesture recognition, and audio triggering, synthesis, and control. The types of sensors that would be employed included accelerometers, gyros, force-sensitive-resistors (FSR's), bend-sensors and potentiometers. These requirements meant that this platform would need to support rudimentary signal-conditioning, multi-channel analog data capture, and wired or wireless serial data transmission.

The Stack [Benbasat, Morris & Paradiso 2003] is a compact, modular, configurable wireless sensing system, for which several sensing boards (e.g., tactile, inertial, sonar, etc.) have been designed. These boards can be stacked in any order and configuration atop a main processor/RF board, allowing the sensor suite to be easily customized. A TDMA polling scheme enables multiple stacks to be used simultaneously. Although it has been primarily designed for wearable applications, this device serves as a general platform for compact multimodal sensing.

The Stack was an appropriate and convenient platform around which to build FlexiGesture. Much of the sensing needs of the device were already supported by the Stack architecture, and additional features were added easily. Two additional modules were built for the stack during the course of the project, an Electric Field Sensing layer, and a user interface-oriented Output layer.

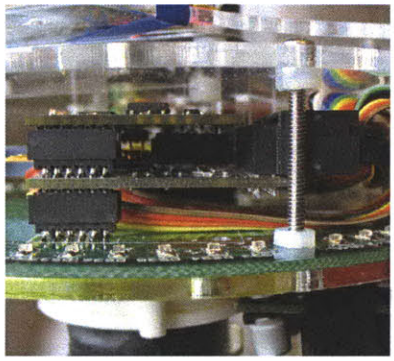

Figure 2-2: Close-up of transmit (top) and tactile (bottom) layers.

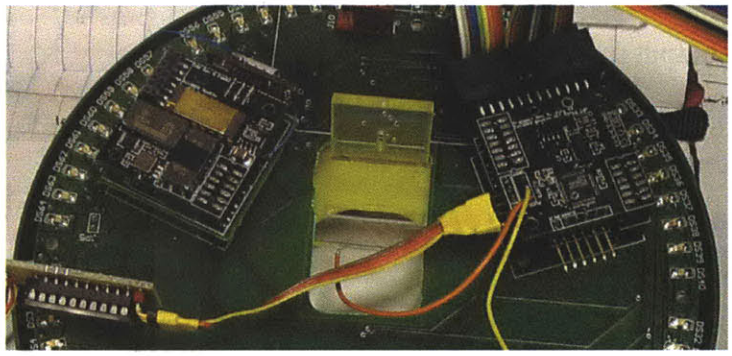

Figure 2-3: Stack components mounted on the circular PCB.

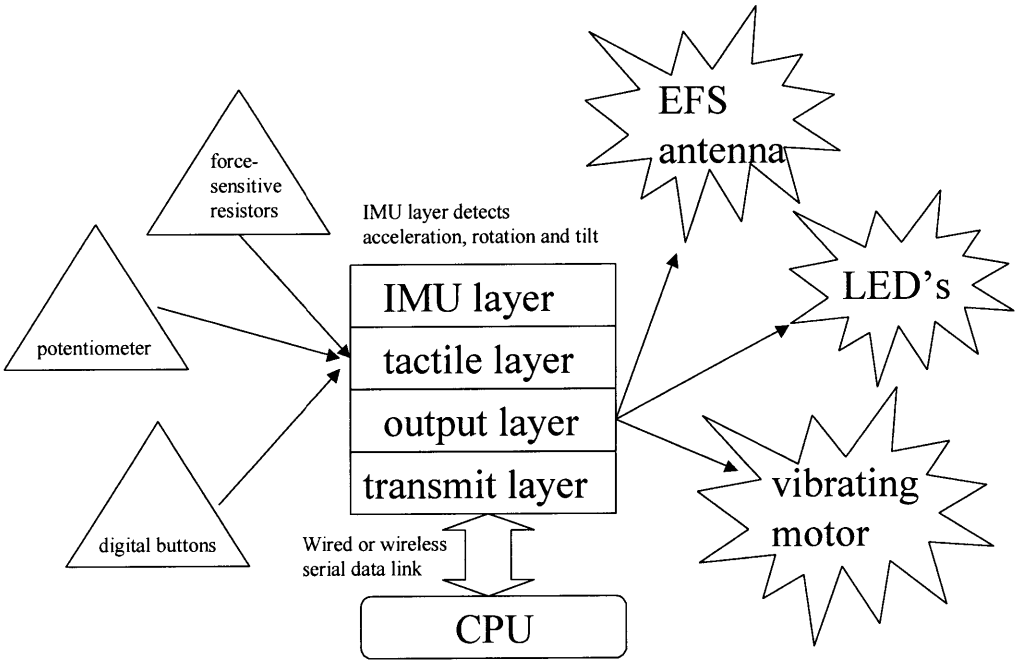

Figure 2-4: Overview of stack configuration used for the device

2.3 Application-specific circuits

The existing stack layers were designed to be a "simple modular framework for wireless sensing" [Stack webpage]. As such, the focus of the development of existing stack layers has been on building data-collection capabilities. With the adaptation of the stack to the current project, the capability to generate in-situ user feedback became necessary. The following sections describe the output layer and the circular PCB, both of which augment the basic stack architecture with feedback capabilities. In addition, modifications made to the tactile layer in order to condition additional FSR and button inputs are described.

2.3.1 Output Layer

The output layer is a printed circuit board (PCB) that fits into the modular stack architecture and that is designed to provide various types of user feedback (see Appendix B for details). The primary modalities of output enabled by the output layer are tactile and visual. Tactile output is generated by a vibrating "pancake" style offset-mass motor that is driven by a NFET that in turn is driven by a pin on the microcontroller. The tactile output modality was not used in the present design, but could be applied to future work.

Visual feedback is made possible by a MAX6951 chip that is capable of independently driving up to 64 discrete LED's. The LED's themselves are mounted on the circular PCB (see section 2.3.2 for details). In addition to the user feedback capabilities, the output layer can generate a 3v peak-to-peak squarewave at 30KHz for use in electric field sensing (EFS). The squarewave is applied to a LC resonator circuit as in the Lazyfish electric field sensing system [Smith 1999], which amplifies the fundamental to roughly 30v peak-to-peak, and this signal appears on a circular electrode built into the circular PCB (see below). The output layer has a Cygnal C8051F331 microcontroller (Now made by Silicon Labs, and hereafter referred to as the 331) onboard that is responsible for driving the vibrating motor and generating a squarewave for EFS operation. The 331's features include an 8051 core, 768 bytes of internal data RAM, 8k bytes of internal flash RAM, 17 port I/O, hardware UART, SMBus, an SPI bus, and an internal 24.5Mhz clock. The 331 and the MAX6951 chip are both connected as slaves to the SPI bus which runs throughout the entire stack architecture. The Cygnal C8051F206 microcontroller on the transmit layer serves as the master on this SPI bus, sending directives to both the LED driver and the 331.

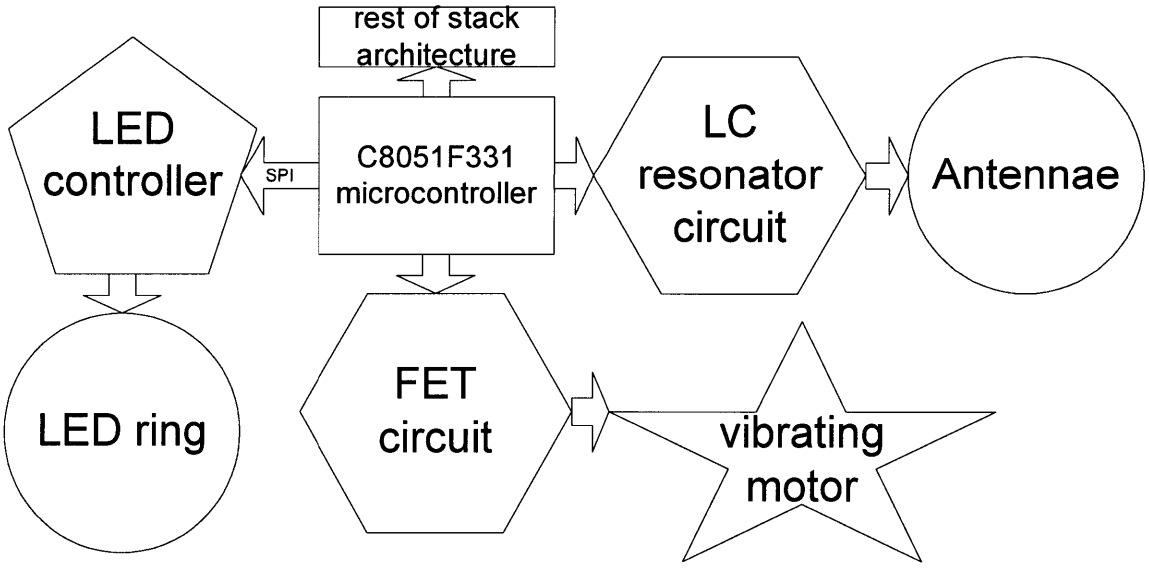

Figure 2-5: Hardware overview for the output layer

2.3.2 Circular PCB

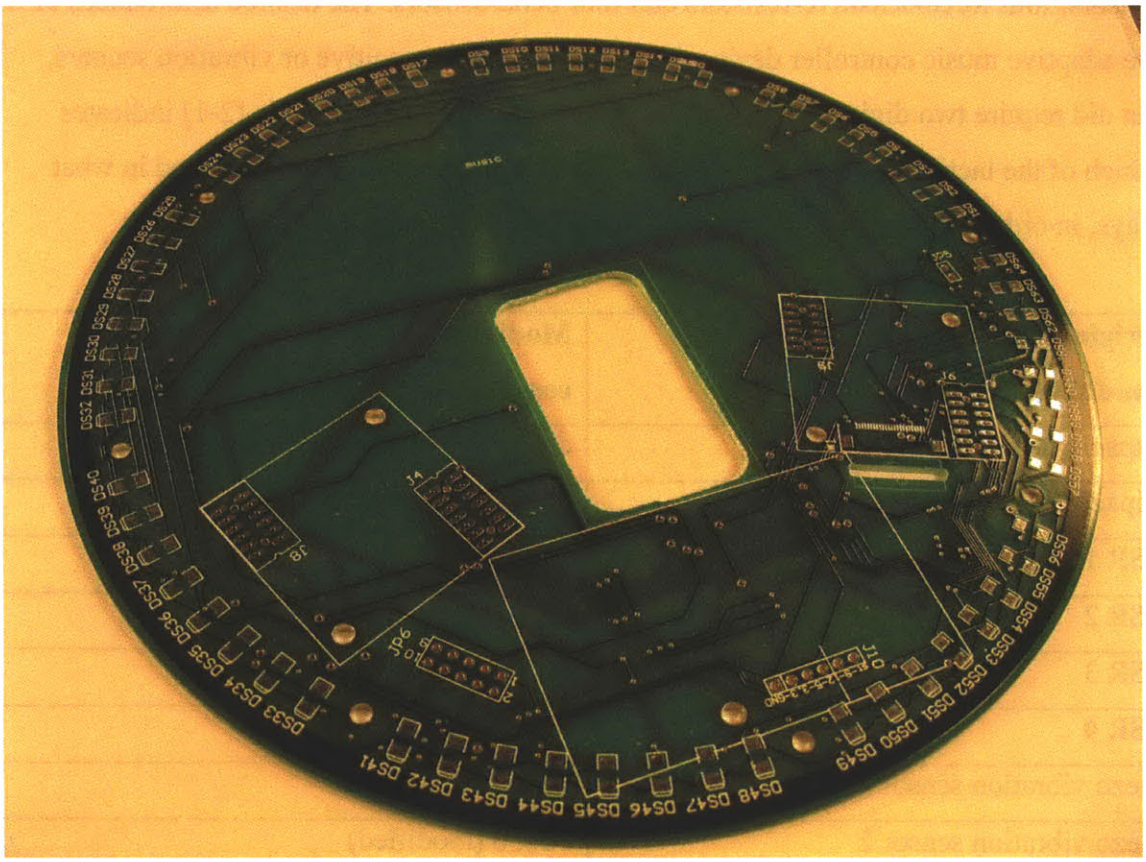

Figure 2-6: The circular PCB

The circular PCB is an expansion board that is used in conjunction with the output layer board. The circular PCB provides footprints to mount the 64 LED's in a circular pattern around its perimeter, as well as a built-in electrode around its edge for the EFS squarewave oscillation. It also has two sets of mounting footprints for the stack, allowing the stack layers to be mounted in two separate places while remaining electrically connected. The circular PCB has eight through-holes around its edge in a pattern that makes it mountable directly on top of a similarly-shaped acrylic layer.

2.3.3 Modifications to the tactile v4 board

The tactile v4 board from the existing stack platform was used as the signal-conditioning and multiplexing front-end for both analog and digital sensor readings. The board was

originally designed to support up to nine capacitive sensors, two piezoelectric vibration sensors, four force-sensitive-resistors and four bend-sensors. The desired affordances of the adaptive music controller device did not require the capacitive or vibration sensors, but did require two digital inputs and two additional FSR inputs. Table [2-1] indicates which of the tactile v4 board's signal-conditioning circuits were modified, and in what ways, in order to provide the needed functionality.

Original Tactile v4 board signal-conditioning circuit	**Modified tactile v4 board signal-conditioning circuit**
capacitive sensor 1	digital button input 1 (modified)
capacitive sensor 2	digital button input 2 (modified)
FSR 1	FSR 1
FSR 2	FSR 2
FSR 3	FSR 3
FSR 4	FSR 4
piezo vibration sensor 1	FSR 5 (modified)
piezo vibration sensor 2	FSR 6 (modified)
bend sensor 1	Bend sensor 1
bend sensor 2	bend sensor 2
bend sensor 3	bend sensor 3
bend sensor 4	bend sensor 4

Table 2-1: Use and modification of signal-conditioning circuits in the Tactile layer

See Appendix B for a complete description of the modifications made to the tactile v4 board.

2.4 Microcontroller embedded code

The primary responsibilities of the embedded code on the C8051F206 microcontroller are to digitize and collect data from the analog sensors, transmit the data back to the PC, and

to generate in-situ feedback for the user. The complete code is found in Appendix C and figure [3-6] shows the data-collection flow.

2.4.1 Data collection

The data collection system runs as fast as allowed by the latencies required by the ADC and serial transmission of the data. On each data collection cycle, thirteen 12-bit samples and two digital values are recorded. The 12-bit samples are taken from the accelerometers, gyros, FSR's and bend sensors. The two digital inputs from the digital input buttons are read directly from port pins on the microcontroller. The entire data collection cycle takes 1.058 milliseconds.

2.4.2 Data transmission

Once the sensor and button data has been collected, it is transmitted by the microcontroller to the PC using the hardware UART. Each 52-byte packet is made up as follows:

number of bytes	**description**
8	header
12	accelerometer and gyro readings
20	FSR, potentiometer, bend and button readings
10	footer
2	'\r' '\n' for line-break
Total bytes per packet: 52	

Table 2-2: Byte allocation in a single stack data packet

The on-board ADC of the transmit layer has 12-bit resolution, meaning that an instantaneous sensor data reading cannot fit into a single 8-bit byte for transmission back to the PC. For this reason, each sensor reading is spread across 2 bytes for transmission, with the least significant 6 bits transmitted in the first byte, and the most significant 6 bits transmitted in the second byte. This explains why, for instance, the 6-DOF inertial data (accelerometers and gyros) requires 12 bytes to transmit. The same rationale holds true for the FSR, potentiometer and bend sensor readings. The state of the two digital buttons

could have been represented efficiently by combining them into a single byte, but for consistency with the rest of the data, the button's instantaneous states were each transmitted in a separate byte, with a 1 or 0 in the least significant bit indicating their state.

The header and footer of a stack data packet are static byte sequences used for padding the data. These sequences of bytes also enable the software that parses the byte-stream to assemble complete data packets to detect the beginning of the sensor data.

For wireless transmission, the sensor data stream must be DC-balanced in order to ensure good receiver tracking of the threshold point [Anthes]. Although the current system does not use the wireless capabilities of the stack, the data was nonetheless DC-balanced in order that the system could be easily used wirelessly in the future. DC-balancing the byte-stream means that each transmitted byte must contain an equal number of 1's and 0's, allowing the receiver to determine a threshold to distinguish 1's from 0's by low-pass filtering the incoming data. Since each of the 12-bit instantaneous sensor readings is split into two 6-bit values, a 6-to-8 bit balancing scheme was used, meaning that each possible 6-bit data value is mapped onto a balanced 8-bit value using a lookup table. Of the 70 balanced byte values possible, 64 were used in the lookup table, and 6 values were not. Five of the unused byte values were ones which have runs of 4 consecutive highs, which can cause tracking problems, as well as the value 0x55, which is reserved for other purposes [Benbasat 2000]. On the receiving end, rudimentary error-detection is performed by throwing out any packet in which a non-balanced byte is received. This scheme allows for error-detection of bytes in which an odd number of bits corrupted, and in practice was sufficient. True (parity-based) error correction was not necessary.

The data-encoding scheme described above was taken from [Benbasat 2000]. The strengths of the scheme are its low memory overhead (64 bytes for the lookup table), its conversion of a 12 bit quantity into two bytes (which are the atomic units of serial data transmission), its guarantee that each byte will be DC-balanced and that there will not be runs of greater than six. Finally, single-bit errors will always be detected.

The entire data transmission cycle takes 4.502 milliseconds, meaning that the period of a single data collection/transmission cycle takes 5.56 milliseconds, for an overall sampling frequency of ~180Hz. The device transmits continuously, except during gesture classification when the PC puts it into no-transmit mode (see section 3.2.3.2 for details).

2.4.3 Feedback generation

The LED's around the perimeter of the circular PCB are used as feedback. The LED's indicate to the user that the trigger button is depressed, meaning that the device is "listening" to the current gesture-in-progress. Once per data-collection/transmission cycle, the microcontroller checks the analog value of the voltage on the right-hand index-finger "trigger" FSR. If the value there is below a threshold, the FSR is considered to be depressed by the user's finger, and all of the LED's are turned on. If the value is above a particular threshold, the FSR is not considered to be depressed, and the LED's are turned off. Hysteresis is implemented in the microcode to prevent flutter if the voltage is near the boundary.

2.5 Form factor and affordances

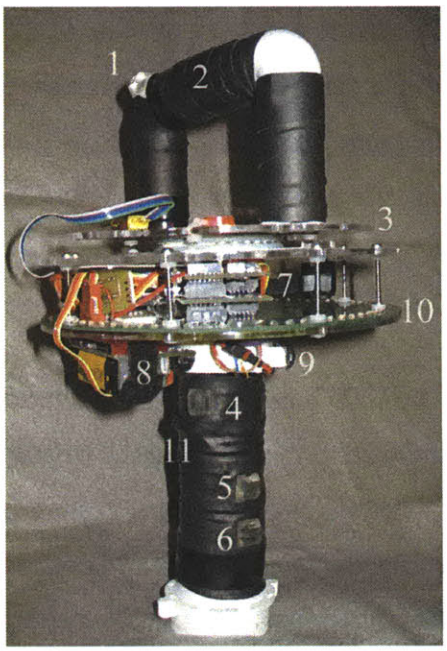

Figure 2-7: The device.

Numbers are indicating the following (1) upper thumb button, (2) upper squeezable handle (3) rotating carriage (4) trigger button (5) lower handle button A (6) lower handle button B (7) inner carriage (8) battery (9) toggle button (10) circular PCB (11) lower handle

The decision to make open-ended gestural control a primary interaction modality informed not only the sensors that were chosen for use in the device, but also the physical affordances implemented. In terms of sensors employed, the accelerometers, gyros, and electric field sensing enabled the use of free-gesture movement of the device as an input to the system. But free-gesture systems are known to be poor input devices for precise control [O'Modhrain 2000]. The theremin, perhaps the most famous open-air noncontact controller, does allow for extremely expressive control, but is so difficult to master that there have been only a handful of virtuosos over the nearly 100 years since its invention in 1919. Don Buchla's lightning is another free-gesture non-contact controller which has been used expressively, but which has not entered the mainstream [Buchla and associates website]. Max Matthews' radio baton [Boulanger & Matthews 1997] has been used for discrete, symbolic control tasks such as providing a tempo for pre-sequenced music as well as for free gesture, expressive pieces. To mitigate the difficulty in achieving precise control typically associated with non-contact gestural controllers, the current device is built also to permit several channels of two-handed, contact-based continuous manipulation by a user. The decision was made early in the design process to permit two-handed manipulation, since humans are adept at performing fine manipulation with both hands, and most musical instruments involve both hands holding or manipulating a single object. Table 2-3 on the next page lists the input degrees of freedom supported by the device, and identifies the type of affordance provided by each sensor.

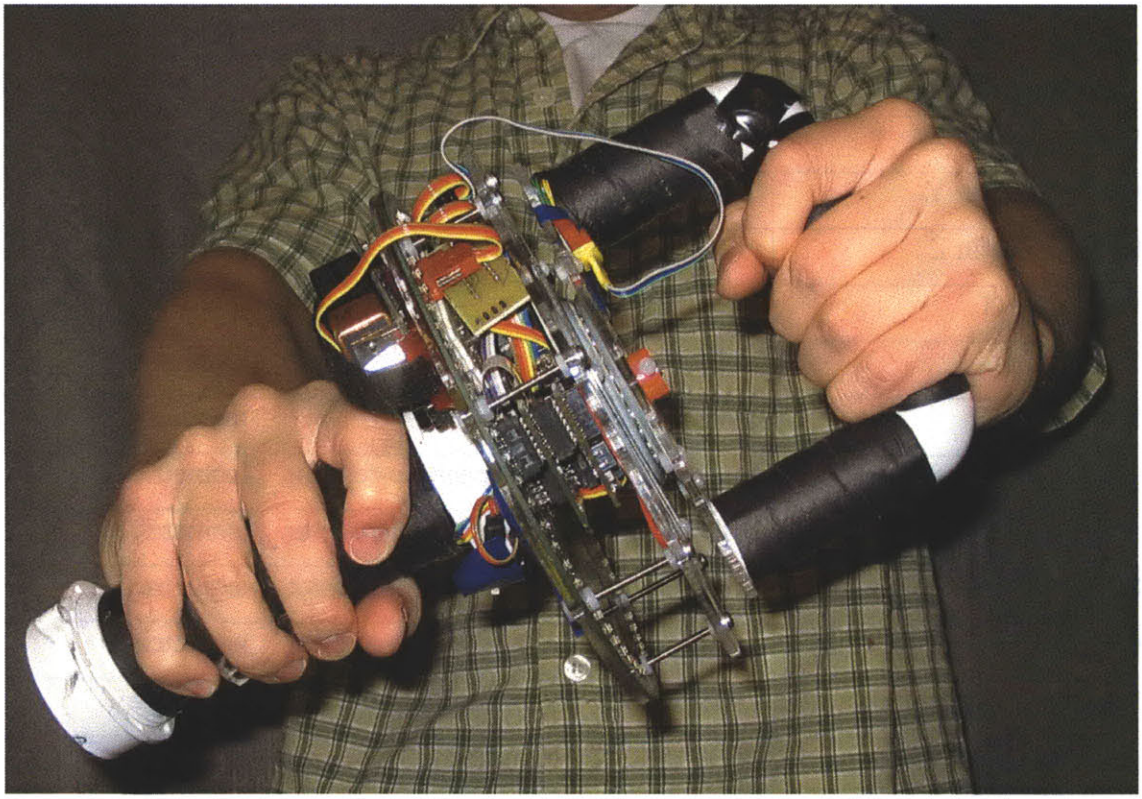

Figure 2-8: The physical device being played

Input affordance	sensor	affordance type
Acceleration in X	ADXL202JE accelerometer	inertial
Acceleration in Y	ADXL202JE accelerometer	inertial
Acceleration in Z	ADXL202JE accelerometer	inertial
Rotation in X	ENC03J gyro	inertial
Rotation in Y	ENC03J gyro	inertial
Rotation in Z	ADXRS150 gyro	inertial
Left-fingers pressure	Interlink 0.2" force-sensitive-resistor	isometric
Left-thumb pressure	Interlink 0.2" force-sensitive-resistor	isometric
Right index-finger pressure	Interlink 0.5" force-sensitive-resistor	isometric
Right third-finger pressure	Interlink 0.5" force-sensitive-resistor	isometric
Right fourth-finger pressure	Interlink 0.5" force-sensitive-resistor	isometric
carriage rotation	1/2W 50K potentiometer	isometric
right-hand handle bend in X	Images SI Flex Sensor	isometric
right-hand handle bend in Y	Images SI Flex Sensor	isometric
up-toggle button	digital button circuit	digital
down-toggle button	digital button circuit	digital
Position in X	Electric Field Sensing circuitry	isotonic
Position in Y	Electric Field Sensing circuitry	isotonic
Position in Z	Electric Field Sensing circuitry	isotonic

Table 2-3: Input affordances of the device

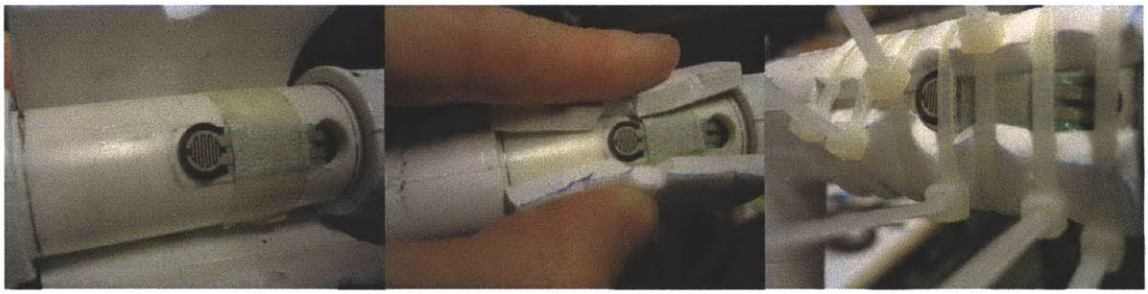

Figure 2-9: How the upper handle's FSR was attached and surrounded with foam

The main body and rotating carriage sections of the device are built with laser-cut 1/8" clear acrylic. The left-hand handle is made of PVC pipe and is attached to the rotating carriage with PVC cement. Informal bond-strength tests found PCV cement and acrylic cement both satisfactory in joining PCV to acrylic (clear epoxy was weaker than either). The left-fingers-pressure FSR is attached directly to the PVC handle, and a rubber foot attached to a length of spring-steel applies pressure to the FSR when the steel is squeezed[2]. A foam strip curled around the handle protects the user from the sharp edges of the steel and diffuses squeezing pressure more equally across the steel. The left-thumb-pressure FSR is mounted directly to one of the 90-degree angle portions of PVC. A rubber foot is attached to the top of this FSR to make it a visible, feel-able affordance. Holes drilled in the left-handle PVC allow connecting wires from both of the FSR's to be routed back to the stack inside the PVC handle and then through a ribbon cable to the main body.

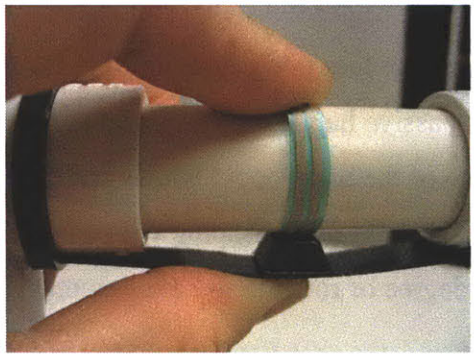

Figure 2-10: Demonstrating the spring steel, rubber foot and FSR mechanism

[2] In the final version of the device, the spring steel is rotated up 90 degrees from the view in figure 2-10 and the FSR is oriented in parallel to the direction of the PVC rather than orthogonally as shown.

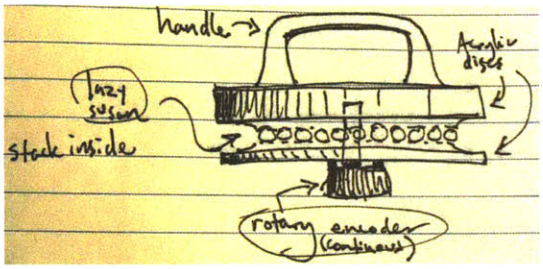

Figure 2-11: Early sketch of the lazy susan idea

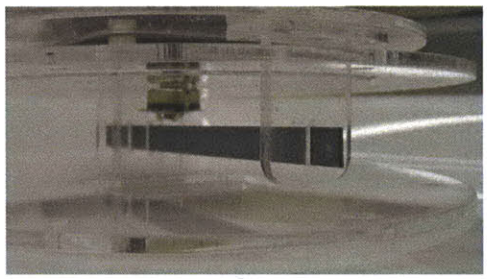

Figure 2-12: The spring steel protrudes through the upper-carriage "tongue"

The upper carriage is attached to the main body with a "lazy susan" rotating platform with ball bearings inside. A vertical tongue attached to the carriage protrudes down through a semi-circular slot in the top of the main body. The tongue serves two purposes: First, it provides a hard stop for the rotation of the carriage to protect the potentiometer from mechanical damage from over-rotation. Second, spring steel attached to the main body protrudes through a slot in the tongue and provides rotational resistance corresponding to the amount of twist given to the carriage. This push-back gives a degree of "passive haptics" to the twisting affordance. A potentiometer is mounted on the upper plate of the main body, which protrudes through the upper carriage. The potentiometer's shaft is kept from slipping with a nylon bolt and thus it turns with the carriage, allowing its degree of rotation to be measured.

The upper and lower acrylic plates of the main body are separated by 1 1/8". The height of the circular PCB is about 1/16", which leaves 1 1/16" of vertical space from the surface of the circular PCB to the bottom surface of the upper main body plate. This clearance is enough to allow for two stack layers to be mounted on each of the mount points. The most height-effective configuration of stack layers featured stacking the tactile and transmit boards together on one mount point, and the output and IMU boards together on the other.

The power board was originally intended to attach directly to the circular PCB. However, when assembling the system it became clear that there was not enough space in the intended area. The ribbon cables coming from the tactile v4

board and the serial line-level converter board both ended up in the space where the power layer was to be. To solve the problem, the power layer was mounted underneath the main body with a velcro strap attached to a custom acrylic harness.

Figure 2-13: The power layer attaches to the bottom of the main body.

The right hand handle is made from a ribbed, flexible piece of plastic tubing. It screws into the bottom of the main body, and acrylic mounting brackets on the inside hold 4 bend sensors in place. The mounting brackets for the bend sensors are slotted to allow the sensors to bend in the direction that they are designed for, and to slide in the other, remaining flat. Three FSR's are mounted to the outside of the right hand handle, placed to rest under the first, second and third fingers. A DB9 connector at the bottom of the handle connected to the transmit layer through a serial line driver IC allows for optional "wired" operation of the device.

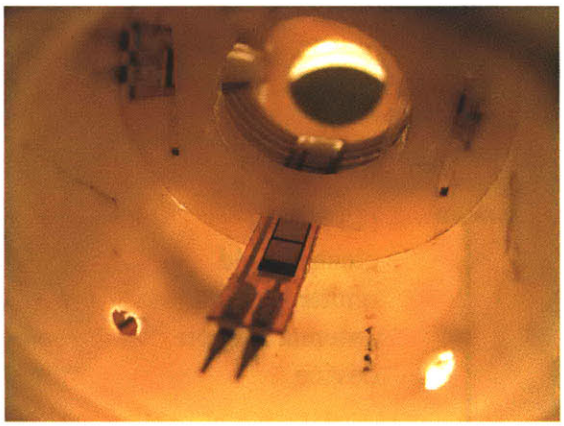

Figure 2-14: Bend sensors mounted inside the right hand handle.

2.6 Electric Field Sensing

The system was designed to support transmit-mode electric field sensing [Paradiso & Gershenfeld 1997] to permit the calculation of the absolute position of the device in a small volume of space in front of a performer. As was mentioned earlier, the output layer contains a LC "tank" circuit which can generate a high-gain voltage

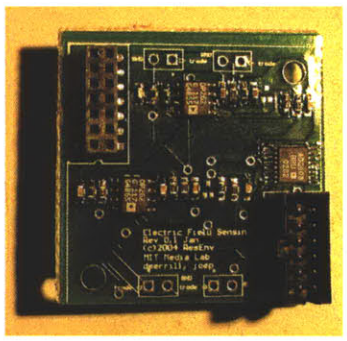

Figure 2-15: The Electric Field Sensing layer

oscillation which, when connected to a suitable electrode, forms the transmit part of an electric field sensing setup. The circular PCB was fabricated with a circular trace around its edge, to be used as the transmit electrode in an EFS setup. The Electric Field Sensing stack layer is made to modularly integrate into a stack base-station. The EFS layer supports the attachment of 4 receive electrodes that pick up the electric field generated by the transmit antenna on the circular PCB. Separate signal-conditioning circuitry is implemented for each of the four receive channels on the EFS layer. The EFS layer was not used as an input DOF during the user study, but will be enabled in future work.

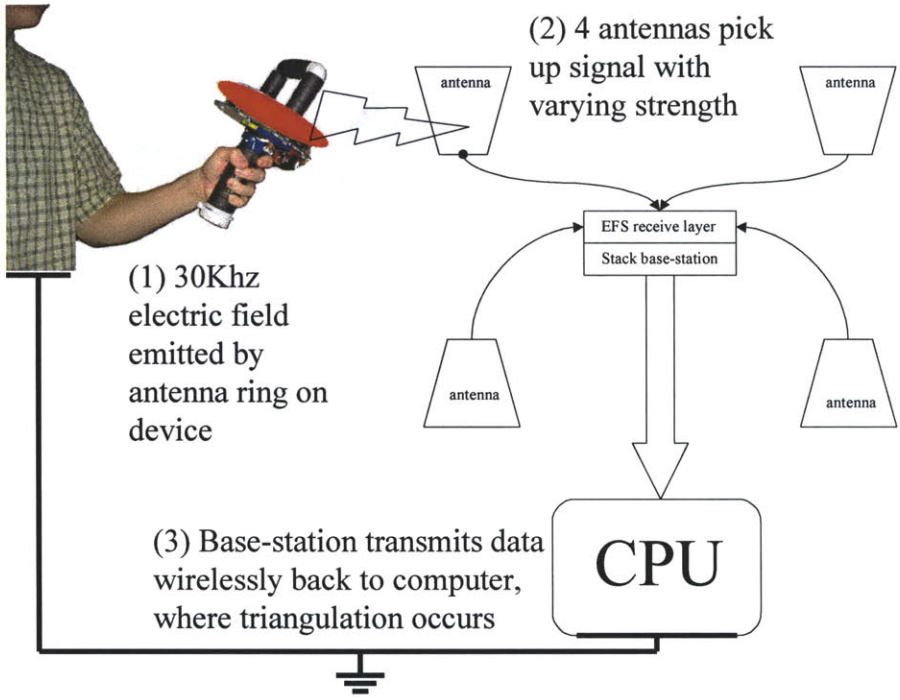

Figure 2-16: Electric Field Sensing setup

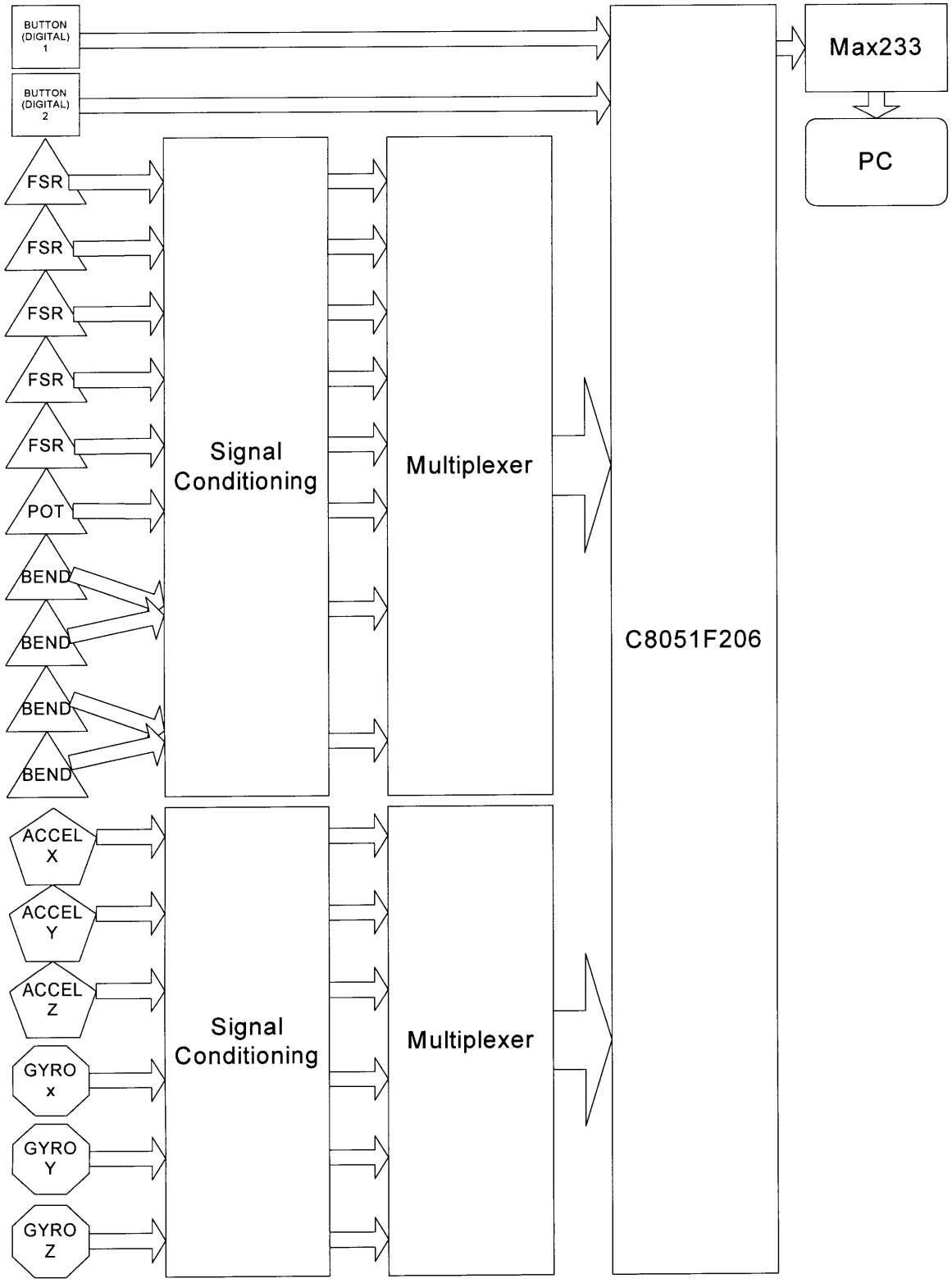

Figure 2-17: Hardware system data-capture overview

Chapter 3

Software: Learning, mapping, and glue

The software component of the adaptive music controller is responsible for allowing the user to explore the sound space, to train gesture-to-sound associations, to create mappings from input degrees of freedom to effect parameters, and to play with the finished result. During play, the software handles incoming data from the controller, performs gesture recognition to retrieve trained associations between gesture and sound, and manages the mappings from the continuous controls of the device onto the sound modification parameters.

3.1 Summary and goals

With the goal of implementing a new paradigm for mapping the affordances of a physical musical controller to output sound, there were a number of software-related goals which all needed to be simultaneously satisfied. The primary goal was that the system present a natural and easy-to-understand way for a user of the instrument to discover, play and manipulate sounds that they were interested in. With the aforementioned as a high-level goal, a number of more specific objectives became clear. These specific objectives included system stability, an easy-to-understand user interface, effective sound space navigation, real-time gesture classification, and a novel assignment scheme of affordance-to-parameter mappings. The following chapter explains the development of the software, and relates the design decisions made back to this set of objectives.

3.2 Java, C, and the JNI framework

Choosing a software framework to use was a non-trivial decision. Choice of programming framework affects the types of data structures and I/O that are convenient to implement and maintain, how a graphical user interface (GUI) can be built, and the overall execution speed and efficiency of the program. A decade ago there would have been fewer options. Constructing a program of this scope would have taken much longer, and included more compromises. Fortunately, in today's software development environment, there exist myriad possibilities for implementation of a project such as this. The combination of tools initially chosen was the following: The GUI and "glue" interface code was written in Java/Swing, while the data storage and pattern recognition routines were initially implemented in C. The Java Native Interface (JNI) API was used to connect the Java code to the C code. For the user study, the C module was put aside in favor of an easier-to-maintain pure Java implementation.

3.2.1 Java graphical user interface

The graphical user interface allows the user of the system to configure I/O settings related to serial input, OSC and MIDI output, to visually inspect the data being collected from the sensors, and to manage the creation of gesture-to-sound mappings and continuous-control-to-effect-parameter mappings. The GUI was created with the free version of Borland JBuilder, a robust and full-featured Java IDE and debugger. The GUI is organized in a tabbed layout style, and each tab will be discussed in turn.

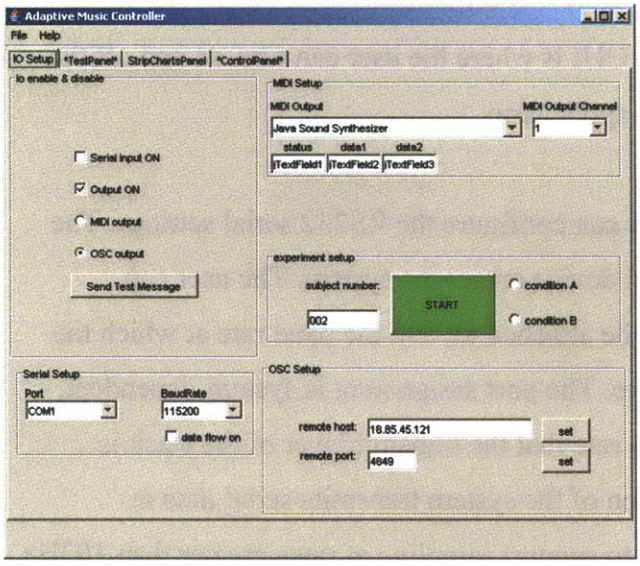

Figure 3-1: I/O setup panel of the GUI

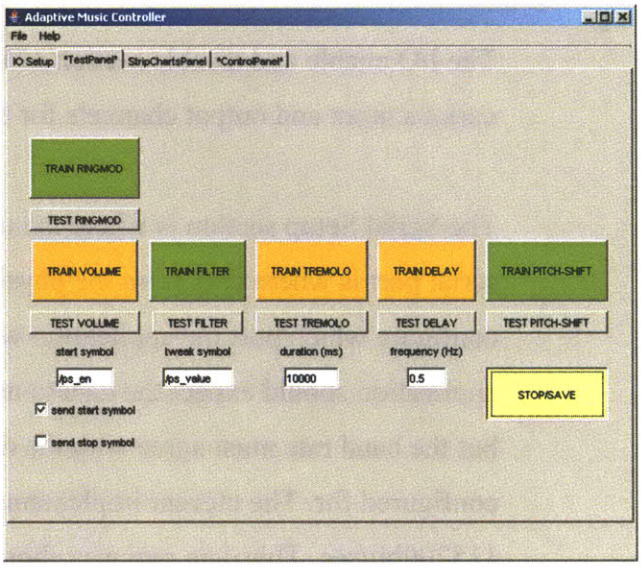

Figure 3-2: Effects mapping panel of the GUI

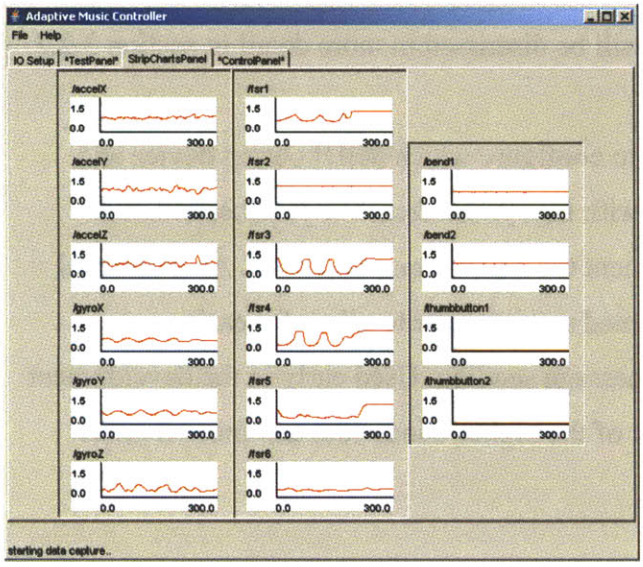

Figure 3-3: StripCharts panel of theGUI

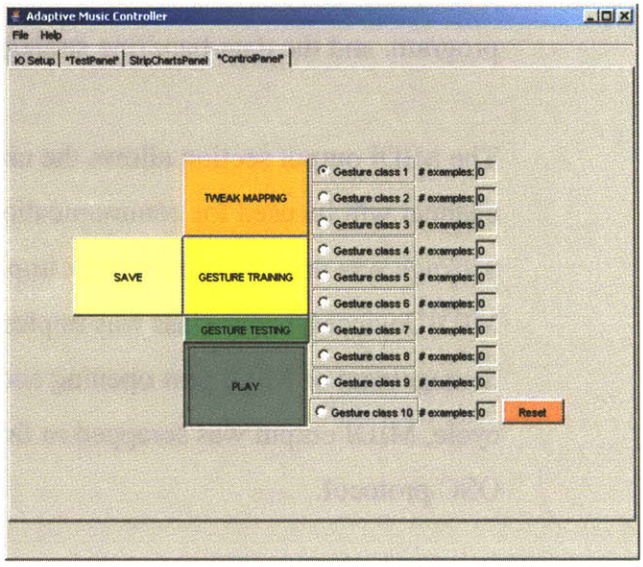

Figure 3-4: Gesture panel of the GUI

3.2.1.1 I/O: For configuring I/O setup

The I/O enable and disable section of the GUI is where the user can turn on and off the various input and output channels for the application.

The Serial Setup section is where the user can configure the RS232 serial settings. The serial port is where data from the physical device enters the system. The user can configure which port the application will be attached to, and the baud rate at which the application should expect the data to arrive. The port assignment is system-dependent, but the baud rate must agree with the data rate that the transmit layer of the stack is configured for. The current implementation of the system transmits serial data at 115200bit/sec. This data rate was chosen to support sampling at rates greater than 100Hz. The Java communications API was used to implement the serial data input to the program, and the data-handling strategy will be discussed in more detail in section 3.2.2.

The MIDI output section allows the user to configure which MIDI output device and channel will be used for communication with the synthesizer. The javax.sound.midi package was used as a framework implement the MIDI communication. An additional MIDIDeviceManager class was implemented as a wrapper to allow for easier management of MIDI port opening and message sending. Used early in the development cycle, MIDI output was scrapped in favor of the higher-bandwidth and more flexible OSC protocol.

The OSC output section allows the user to configure which IP address or hostname, and to what port OSC datagram packets should be sent. OSC is a protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology. A JavaOSC library from Illposed Software was used to implement the OSC functionality [Illposed Software webpage].

The experiment setup section was created to facilitate the running of the user study. This part of the GUI allows the experimenter to assign a unique identifier to the current

subject, to specify to which experimental condition the current subject belongs, and to start and stop data logging for the current subject. The software creates a set of directories where the data for the current subject is written.

3.2.1.2 TestPanel: For creating input-DOF to effect mappings

The TestPanel tab in the GUI allows the user to create input-DOF-to-effect mappings. It has a button for training each available effect, and a separate button for testing each effect. A Stop/Save button stops any ongoing data capture and saves the current configuration to disk.

3.2.1.3 Stripcharts: For data visualization

Software and hardware oscilloscopes are a convenient way to visualize real-time signals. The StripCharts panel of the GUI allows for the simultaneous, real-time visualization of the 16 sensor/button data streams coming from the device. Each individual StripChart is a Java component subclassed from JPanel. The StripChart shows a line-graph that scrolls from right-to-left, updating at the rate that the data arrives from the hardware. The original StripChart code was written by Andy Wilson [Wilson 2000], and was modified by the author in order to allow the user to toggle the drawing behavior, sample storage and variance detection on or off. In practice, the author tends to enable related batches of charts, such as all accelerometers at once, or all gyros, in order to visually inspect their functionality. This enabling/disabling feature for the StripCharts was necessary in order to manage the CPU bottleneck on the Swing (graphics) thread produced by multiple StripCharts all attempting to update at the ~180Hz data rate. The amount of data history retained by a single StripChart was set to 100 samples. This value was tuned by the author to balance usefulness with CPU load, since the time required for the drawing of each StripChart increases linearly with the number of points stored.

3.2.1.4 Gesture control panel

The gesture control panel permits the user to listen to the available sounds, and to make gesture-to-sound associations. In Training mode, any gesture executed will be associated with the most recently played sound. In Testing mode, any gesture executed will be classified against the stored models, and the appropriate sound will be triggered. Play mode is like Testing mode, except that the input-DOF-to-effect mappings are also enabled.

3.2.2 C and Java modules for data storage and manipulation

In order to achieve the fastest possible execution time for the gesture recognition algorithms, the decision was made initially to store and analyze the data in a compiled C dynamically-linked-library (DLL) rather than in the Java environment. Compiled C has historically been faster than Java, although benchmarks show that Java is closing the gap, arguably even outpacing C on certain tasks. The Java Native Interface (JNI) was used to write a C DLL that can be loaded into the Java environment. The C module has functions for accepting packets of sensor data, storing the data, computing class models and computing similarity scores between novel gestures and class models. Section 3.5 explains the gesture recognition in more detail. Shortly before the user study was to be run, a pure-Java version of the data storage and manipulation module was written and substituted into the codebase in place of the C DLL. This switch was made in response to difficulty in tracking down a latent bug the C module. The pure-Java module was easy to maintain, but did increase gesture-recognition latency (see table 3-1 for a comparison).

3.2.3 Multi-threaded serial data handling framework

In order to guarantee the availability of the data-receiving thread while handling the large quantity of incoming serial data, the following scheme was used: A queue in the SerialConnection object is filled with incoming bytes of serial data. This queue is also used as a semaphore, with access to the queue limited to either the SerialConnection object, or a SerialGrabber object, but always no more than one at a time. Each time the

serialEvent callback in the SerialConnection object is called, indicating that new serial data is available, the new bytes are inserted into the queue, and notify() is called, freeing up the semaphore. This allows the SerialGrabber object, which is waiting on the semaphore, to dequeue all of the new data, inserting the bytes into its own separate queue, then immediately freeing the semaphore. Once outside of the synchronized code section, the SerialGrabber object notifies any listeners that are registered to accept new serial data. The minimal cycles spent inside the synchronized code region by both objects ensures efficient data handoff between them and the smoothest possible reception of the data.

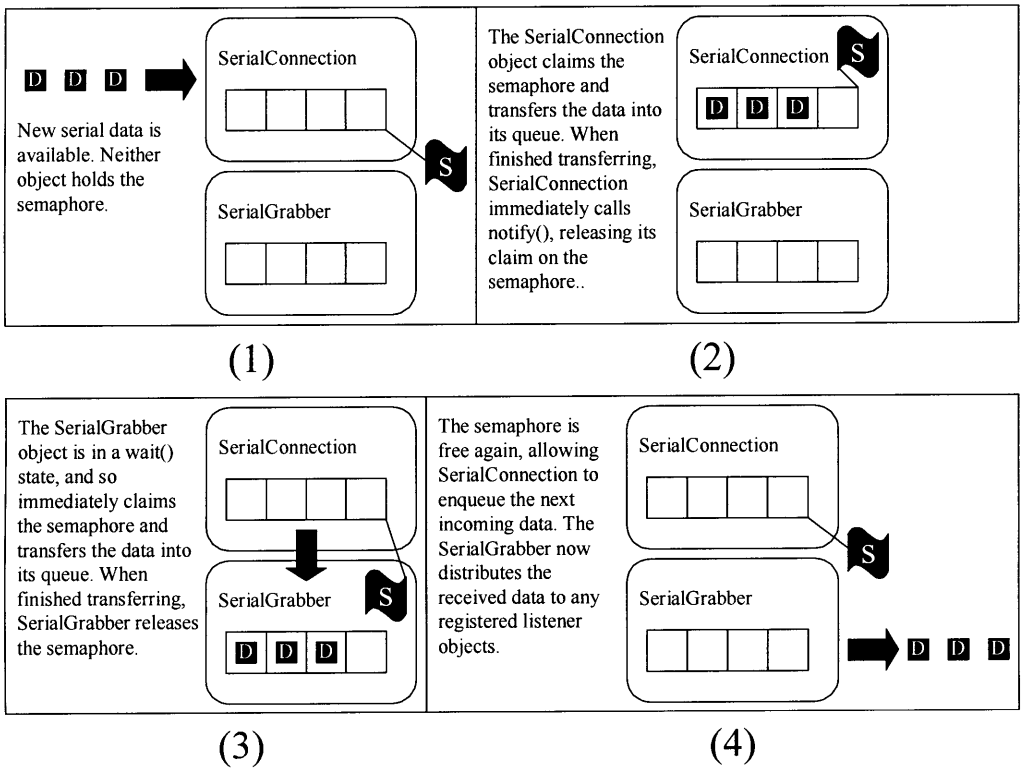

Figure 3-5: Semaphore-based serial data handling

3.2.3.1 Listener structure

Listeners are software objects that are registered to receive notification about events in a program. The use of listeners is typical in an object-oriented programming language like Java. In the current application, data from the serial port is handed off to a StackDataFilter object, which is responsible for assembling incoming data into a

complete packet of sensor data values. Each packet contains all of the values from a single sensor data collection cycle in the microcontroller. Once assembled, the packet is passed off to a number of other listener objects for storage, display in the GUI, and the triggering of navigation and gesture-recording start and stop functions.

3.2.3.2 Managing the serial data flow during CPU-intensive operations

During testing, it was found that inflow of sensor data could crash the system if the application was not in a state to actively receive the data. This behavior was traced to a bug in the USB serial port emulation driver on the author's computer. The problem would typically happen during gesture-recognition, an operation that was so CPU-intensive that the rest of the program would suspend temporarily, meanwhile serial data would continue to stream into the system. At times, this pause in responsiveness was long enough for an internal driver buffer to overflow, crashing the machine. The successful workaround was for the application to ask the stack to stop sending data during gesture recognition. To accomplish this, bidirectional communication between the computer and the hardware was enabled, and just before gesture recognition begins, the java program sends a 0x00 byte to the stack, putting it into no-transmit mode. After recognition completes, a 0x01 byte is sent, re-starting the flow of data to the PC. This loss of sensor data values during gesture recognition was deemed an acceptable compromise, since the time duration of the recognition process was small (see table 3-1).

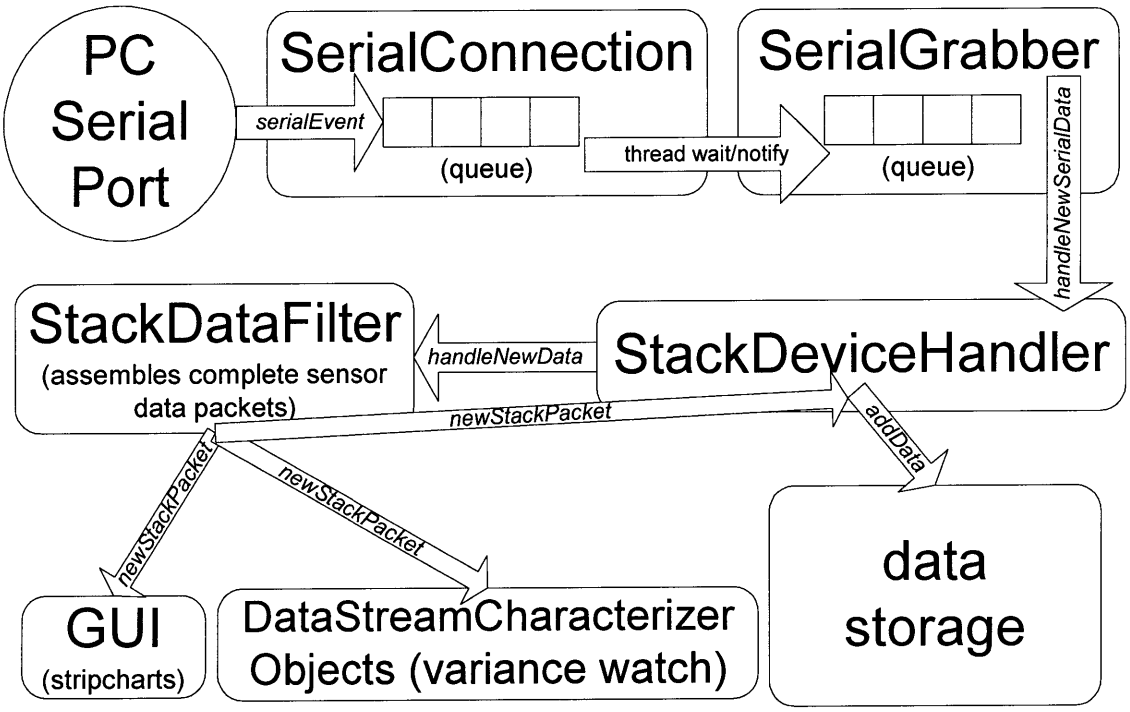

Figure 3-6: Software (Java) handling of incoming serial data

3.3 Trigger and Modify

In planning out the interaction style for the device, a conceptual framework was needed to guide the affordances, both physical and software, that would be implemented. The high-level goal for the device was that it would begin its relationship with the user as a blank slate, and would learn to respond appropriately to the user's actions through a training interaction. Moreover, in order to be as flexible as possible in the kinds of sounds that could be produced, it was determined that the interaction should include an element of symbolic "selection", as well as an element of continuous "sculpting". The idea behind the symbolic selection aspect of the interface is that it would allow random access to a wide space of "ingredient" sounds, while the continuous sculpting part would permit these sounds to be warped in expressive parametric ways. In order to implement a system, this goal had to be turned into a concrete specification, which raised a number of questions. Should the sounds come from an algorithmic synthesizer, or from stored digital samples? Should changing the timbre of a sound consist of a parametric variation in the underlying synthesis technique, or should it modify the spectral content of the

already-synthesized sound? The resolution of these questions, and others, resulted in the following high-level system design:

The conceptual model for playing the system can best be described as "trigger-and-modify". What this means is that, when playing the system, the typical interaction cycle is for the user to first trigger a desired sound (the symbolic part), then to "sculpt" the timbre of the sound in a subtractive manner (the continuous part). In order to create an adaptive device, both parts of the interaction model needed to be "trainable" by the user. The system design of these individual components will be discussed in the following sections.

3.4 Exploring the sound space

Digitally sampled waveforms are used in the system as the gesturally-triggered content. Various algorithmic digital audio synthesis techniques were considered, such as granular synthesis, frequency modulation, and physical modeling. In order to explore the parameter spaces offered by these techniques in a satisfactory manner, a compelling parameter-space traversal interface would have been required, the development of which was beyond the scope of the thesis timeline. Playing and looping digital samples was straightforward to implement, and it also avoided constraining the sound of the device to the character of a particular algorithmic synthesis technique. With a set of samples for the user to choose from, a natural way to traverse them was needed. A toggle button on the handle, positioned underneath the user's right thumb, was used to traverse the samples. Conceptually arranged in a linear list, the next sample in the upward direction in the list could be heard by pressing the toggle button up. Similarly, the user could explore in a downward direction by pressing the button down.

Figure 3-7: Pushing the toggle button up

3.5 Gesture recognition

Human gesture recognition is a subset of the larger field of pattern classification. In the abstract, the task of a pattern recognizer is to accept a novel datum, which can be a single point, or a signal represented by a sequence of samples, and to classify the datum into one of a set of known categories. For instance, a speech recognizer is a pattern classifier that operates on time-domain recordings of sampled human speech. The input to a speech recognizer at classification-time could be a few seconds of speech, and the output would be a textual representation of the words contained therein, perhaps along with a confidence score. The confidence score represents how "sure" the algorithm is about its response, and a client program may choose to throw out a classification result if the confidence score falls below a certain threshold.

3.5.1 Statistical pattern-recognition methods

Statistical pattern-recognition methods are a class of pattern classification algorithms that operate by characterizing the variability in the training data, then using this characterization to determine the most likely class of a novel datum. Variability is typically represented as a Gaussian, and in order to train these systems, enough training examples of each class to reasonably estimate appropriate parameters of the Gaussians are needed.

A number of what we will call "vector-based" statistical pattern recognition techniques are popular with researchers today. Support vector machines (SVM's) are an example of this class of techniques, and they require new data to not only have the same dimensionality as the training examples, but it must also have the same number of samples. Given a method for making all incoming data examples the same length, these vector-based techniques can be quite useful. However, in their basic form most of these algorithms don't work well for time-varying data such as gestures, where the number of samples of each new gesture is different.

Hidden Markov models [Rabiner 1989] are a statistical pattern-recognition construct that can handle time-varying signals. The come in two basic configurations, continuous and discrete. The discrete variety assumes a given "alphabet" of states through which data vectors are expected to travel, while the continuous variety is flexible about the number of states, building more or less of them as needed to describe the training data. Both HMM's and other statistical techniques tend to require a substantial number of training examples in order to become robust classifiers, and for that reason they were not used in this work. See chapter 5 for ideas about how HMM's could be useful in future work.

3.5.2 Dynamic time-warping

Dynamic time warping (DTW) is an algorithm based on dynamic programming (DP) [Bellman 1957] that finds the optimal alignment between two variable-length sequences of data, under certain constraints. In finding the optimal alignment between two sequences of data, the dynamic time-warping algorithm warps the two vectors to each other non-uniformly, stretching some regions while compressing others. Dynamic programming operates on the principle that the optimal solution to a large problem should contain optimal solutions to its sub-problems, and that these smaller solutions can be stored rather than being computed at each step. DTW has been used heavily in speech recognition [Sakoe & Chiba 1978] and DNA sequence-alignment tasks [Needleman & Wunsch 1970], and in general is well suited to processing of time-domain sequential data.

In the current implementation, the input vectors differ in length from the length of the model vectors. In contrast to the DNA sequence-alignment and time-domain speech classification tasks, the gestural data vectors are multi-dimensional, representing the six inertial data streams. (in future work, the number of features used in the DTW algorithm will be expanded to include other continuous-control data) To handle this difference during computation, the distance from a single novel "frame" of 6 values to a single model "frame" is taken by a sum-of-squared-differences technique across the 6 data values.

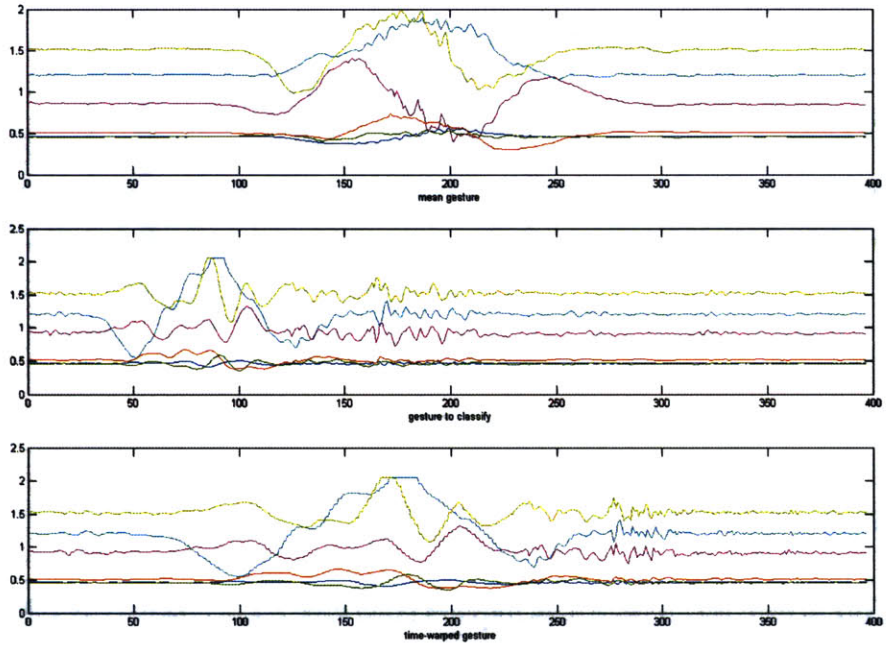

Figure 3-8: Dynamic time-warping in action

The top image is a model gesture, and the middle is a new gesture for classification. The bottom image is the new gesture, warped to show its best fit to the model gesture.

Dynamic time-warping was chosen as the gesture-recognition algorithm for the current project because it provides decent recognition rates in polynomial time, while allowing for model-training with a small number of examples. Most importantly, DTW can operate on data vectors with different lengths. A fundamental usability goal for the system was that it should allow a user to create a gesture-to-sound association easily, and without excessive training. This goal suggested that a gesture-classification scheme should be used that could classify as robustly as possible without a large number of examples of each gesture. The statistical methods mentioned in section 3.5.1 tended to be less suitable given this goal, since they require significant amounts of training data before they become useful classifiers.

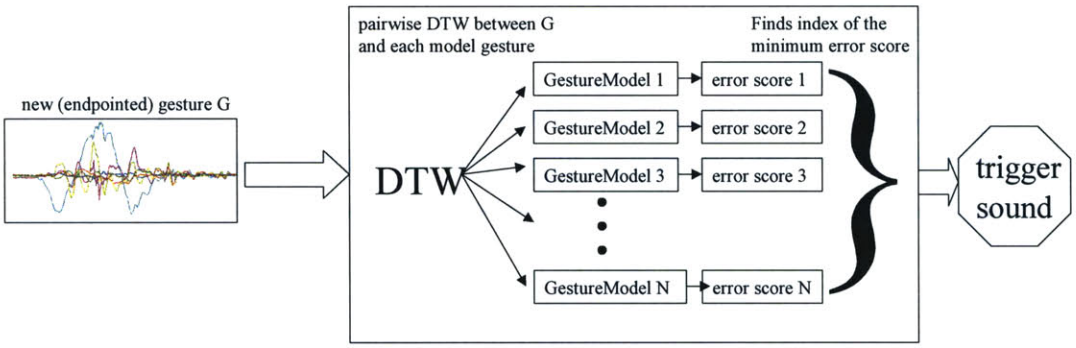

Figure 3-9: Overview of the gesture-recognition process

3.5.2.1 Speed of the DTW algorithm

The dynamic time-warping algorithm as implemented here has a complexity of O(N x M x D), where N is the number of frames[3] in the novel gesture, M is the number of frames in the model gesture, and D is the number of features being used. Since the number of features is typically some small constant (in the current system D is 6, since we are using the 6 inertial DOF's), this complexity reduces to O(N x M). Practically speaking, what this means is that, as the length of the gestures get large, the algorithm requires an amount of time that is proportional to roughly the square of the length of the gesture. In addition to time, the space required for the algorithm to run is also O(N x M). Following is a chart listing some sample classification times measured on the system (these measurements were taken with the Java Virtual Machine (VM) v1.4.2_04-b5 from SUN Microsystems, with 256MB RAM allocated to the VM, on a 2.66GHz Xeon processor host machine with 1Gb of system RAM). No just-in-time (JIT) compilation or other optimization was used, but it is expected that these techniques could significantly speed up the Java execution time in future work. Fortunately, symbolic gestures tend to be short. In the user study (see chapter 4), most of the gestures trained by subjects tended to be less than 1 second in length, and the classification time scales only linearly with the number of model gestures. So with 5 gesture classes trained, and an average gesture length of 750msec, classification in java would take on the order of ~240msec. Additionally, the classification-latency issue is one that faster processors will continue to

[3] A "frame" is taken to be a single time-slice of sensor data. In this case, a single frame of inertial data would consist of 6 values.

alleviate. The continuous effects (see section 3.6) were not timed for latency, but did not require classification, and thus were quite fast.

Short Gesture	
Java	C
Model: 224 frames, 1344 msec New: 214 frames, 1282 msec	Model: 197 frames, 1250msec New: 154 frames, 922msec
Running time: 62 msec	Running time: 0 msec*
Medium Gesture	
Java	C
Model: 537 frames, 3094 msec New: 551 frames, 3218 msec	Model: 532 frames, 3125 msec New: 517 frames, 3000 msec
Running time: 672 msec	Running time: 94 msec
Long Gesture	
Java	C
Model: 2270 frames, 12782 msec New: 2451 frames, 13765 msec	Model: 2292 frames, 12860 msec New: 2317 frames, 13015 msec
Running time: 10453 msec	Running time: 4469 msec

* = Java timer resolution too coarse to detect any time difference

Table 3-1: Runs of the DTW algorithm in C and Java, and associated running times

3.5.2.2 IMU-based dynamic time-warping recognition trial runs

In order to test the feasibility of using dynamic time-warping for gesture recognition, a series of trials was run to test how accurate it could be for the present task. Specimens of the inertial parameters (acceleration, rotation) from 10 distinct gesture classes were recorded. Each specimen consisted of about 360 samples/sensor (~2 seconds of data) from each of the 6 data streams. In total, 30 examples of each gesture were recorded. It was decided that the best way to understand the fitness of various classification algorithms would be to test them in a situation similar to how they would be used in the application. The verification process went as follows: of the 30 gestural examples from each class, 5 were taken at random to comprise the training set and the remaining 25 were marked as test vectors. In order to understand how DTW performed with respect to some other possible pattern classification alternatives, the following schemes were all run through classification trials:

3.5.2.2.1 Naïve distance from the mean gesture:
In this approach, a model gesture of each class was computed by resampling each of the training examples to be the same length, and taking the sample-for-sample mean across the training set. Classification of a novel sensor data stream consisted of finding the model gesture that had minimal distance from the novel data.

3.5.2.2.2 Fourier descriptors:
In this approach, a model gesture of each class was computed by taking the Fourier transform of each of the training examples, removing c(0) (the DC frequency coefficient), and normalizing the rest of the frequency coefficients c(k) by c(1). Then the model gesture was computed as the per-index mean of the first 100 frequency coefficients (ranging from 0 to 50Hz) across the 5 training examples. Classification of a novel sensor data stream consisted of performing the same spectral decomposition as was done for the training examples, then finding the nearest neighbor in the 100-dimensional normalized Fourier coefficient space.

3.5.2.2.3 Pre-processed naïve distance:

This approach was similar to (1), but the data streams (both training and novel examples) were pre-processed in order to trim excess "dead space" before and after each gesture example. The location of the dead space was determined based on observing the typical variance on the sensors during inertial activity versus the typical variance while the device was "at rest". Once the "at rest" had been characterized, the individual gesture

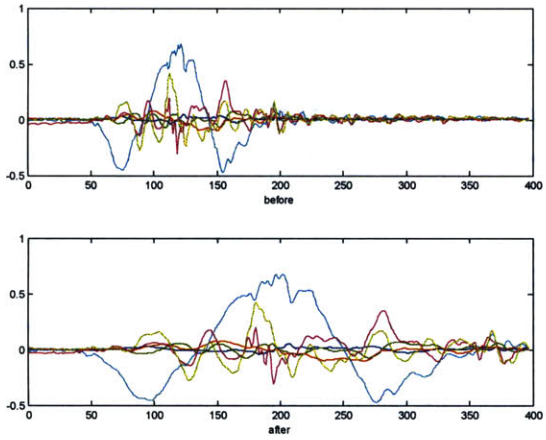

Figure 3-10: A single gesture, before (top), and after (bottom) preprocessing.

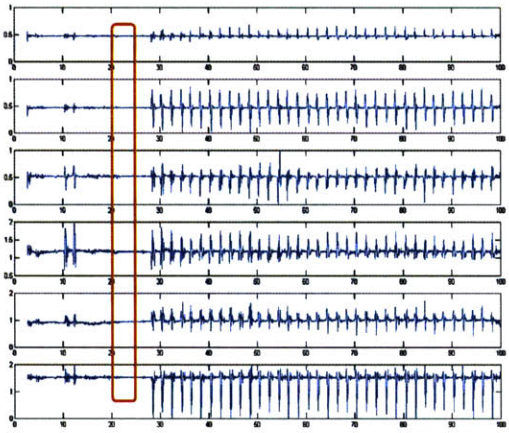

Figure 3-11: Locating a region with typical "at rest" variance levels.

examples could then be trimmed to remove leading and trailing dead-space – segments where the variance was less than a threshold that was a small multiple of the "at rest" variance.

3.5.2.2.4 Pre-processed Fourier descriptors:

This approach is similar to (2), but the data was trimmed as described in (3).

3.5.2.2.5 Dynamic Time-Warping:

In this approach, dynamic time-warping was used on the non-preprocessed data. Model gestures were computed as in (1).

3.5.2.2.6 Pre-processed dynamic time-warping:

In this approach, dynamic time-warping was used on pre-processed data. The data was pre-processed and model gestures computed as described in (3).

In summary, pre-processing combined with the naïve nearest-neighbor method achieved results that were quite good (95%), but dynamic time-warping without preprocessing emerged as superior to any of the other techniques (98%). The fact that the fourier-descriptors technique got worse after preprocessing is understandable since resampling the gestures after trimming (such that they are all the same length) alters spectral content.

	naïve	fourier	DTW
not pre-processed	.75	.90	.98
pre-processed	.95	.82	.97

Table 3-2: Results of the gesture-recognition trials

Due to the good recognition rates found, the dynamic time-warping algorithm was chosen as the gesture-classification technique for the system. The astute reader will notice however, that the pre-processed naïve method achieved classification rates, through lower, were quite close to the rates reached by the dynamic time-warping. Moreover, the naïve method has a basic complexity of O(N), which scales linearly with the length of the data vector. The drawback of the naïve-preprocessed technique would have been the need to trim, then resample novel gestures to a standard length before they could be compared against a class model. It is not known if the use of the pre-processed naïve method would have resulted in faster overall recognition. This question could be investigated in further work, but perhaps even more fruitful would be a look at continuous hidden markov models for a future refinement of the system (see chapter 5 for more on this thought).

In order to easily segment the data stream into individual gestures (either for classification or training), the state of the "trigger" button on the device was used by

software to start and stop data capture. This made easy the task of reliably endpointing a single example, which is a common problem in pattern classification systems that operate on continuous streams of data.

3.6 Interactive mapping of the device inputs

The design goal for the mapping of the input degrees-of-freedom to the effect parameters was to make this assignment intuitive and adaptive. In order to keep the system design tractable, the decision was made to allow the user to create one-to-one mappings, where a single input degree-of-freedom would be linked to a single effect parameter. The input degrees of freedom enabled for this purpose were the squeezing pressure on the upper handle, the pressure on the thumb button on the upper handle, the rotation of the upper carriage with respect to the lower, two directions of bend on the lower handle, and the squeezing pressure on the two buttons on the lower handle. Left for future work is the enabling of the electric field sensing position, and tilt-sensing on the device as additional continuous inputs.

In order to make an input-DOF to effect parameter association, the system sets into motion a 0.5Hz oscillation of the given effect, and simultaneously begins watching the variance level on all of the enabled sensors. To train the association, the user must "follow along" with the oscillation on a given input sensor, actuating it towards one end of its dynamic range when the effect is near one extrema, and towards the other end of the range when the effect is near the other extrema. For instance, if the effect to be trained is a sweeping bandpass filter, the system begins by turning on white noise source and sweeping a band-pass filter effect from the bottom to the top of its range on a period of 2 seconds. Suppose the user chooses the upper-handle squeeze DOF as the input that they would like to assign to the filter effect, and they start to follow along with the effect. They might squeeze hardest on the handle when the filter is at its lowest center frequency, and most lightly when the filter is at its highest center frequency. After a cycle and a half of this (consisting of the passage of 3 extrema), the system notices that (a) the upper handle FSR is experiencing the most variance for the past 3 readings, and that (b)

the "polarity" of the activity on the given sensor is such that a low value on the sensor (tight squeeze) corresponds to a maxima of the filter's center frequency. With three consecutive consistent readings, the software makes the association between the squeeze pressure on the upper handle and the filter effect, with an inverse mapping of squeeze pressure to center frequency. The input DOF associated and the polarity of the mapping are thus flexible, depending on what the user wants, and created by just following along, allowing the user to teach the system "by example".

3.7 Sound Synthesis

Pure-Data (PD) was used for audio output. PD is an open-source free-software implementation of a MAX/MSP dataflow-style application that runs on Windows, Linux, Irix, and Mac OSX. Operating on a "dataflow" metaphor means that samples are generated from a source (in our case they are read out of an array which is loaded from disk), and various digital signal processing (DSP) operators – or "effects" - can be applied to them before they are sent out to be turned into analog audio by the computer's audio hardware. If a parametric audio synthesis technique had been used, the continuous-control element of the interaction could have included modification of synthesis parameters in addition to modification of the effect parameters. The effects used for the current system are a band-pass filter, tremolo, delay, pitch-shift, ring-modulation and volume. The "patch" (as PD programs are called) accepts input in the form of OpenSoundControl (OSC) messages, which can trigger playback of a sample, enable or disable looping of a particular sample, and modulate effect parameters.

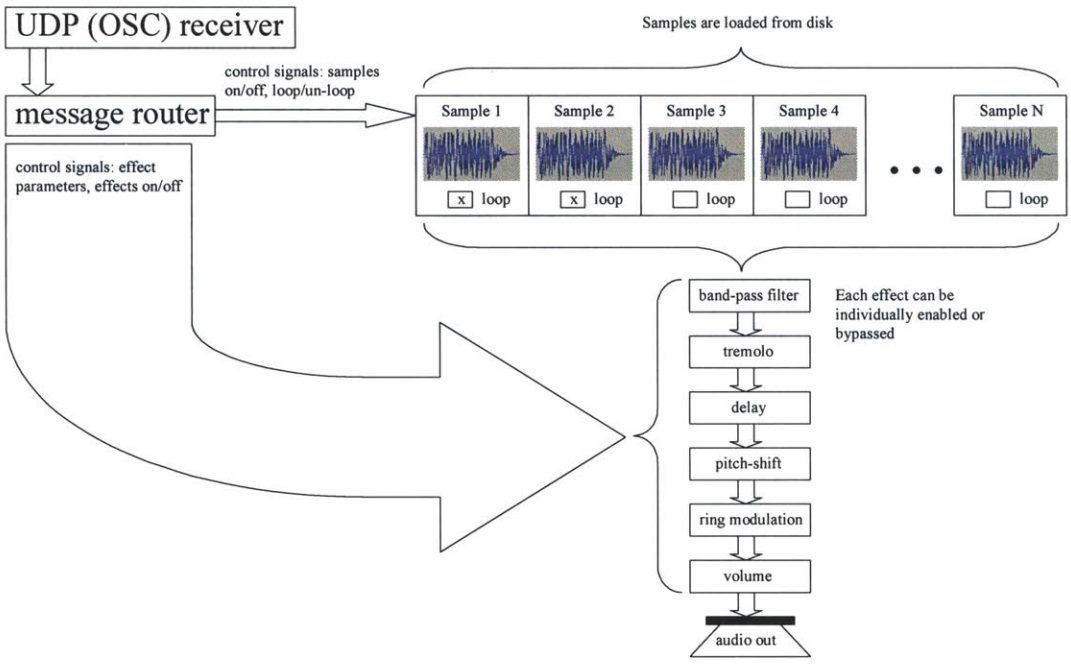

Figure 3-12: Control data (left) and audio (right) flow in the pure-data patch.

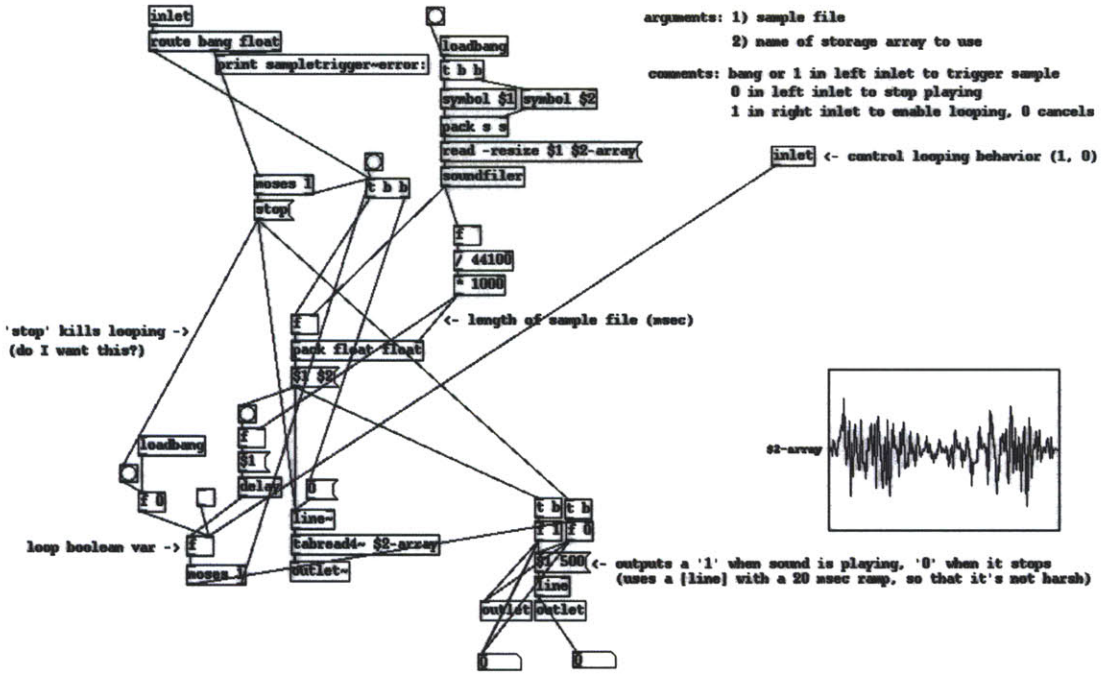

Figure 3-13: The PD patch responsible for loading and playing back an audio sample

3.8 Software conclusions

The system described provides an interface by which a player can navigate the set of available sounds, can assign an open-ended inertial gesture to a sound, can create mappings from a number of continuous input DOF's to effect parameters, and can then play with the result. In future work the gesture space available for triggering of sounds will be expanded to include the contact-based, and electric-field-sensing inputs. The next chapter will describe the user study that was conducted in order to evaluate the adaptive features of the system.

Chapter 4

User Study: Design and results

In order to evaluate the interaction characteristics the new musical interface paradigm embodied by the system, it was important to evaluate it with a user study. There were two main goals of the study: First, explicit feedback about the interaction would be collected from participants about factors like engagement, perception of novelty and personalizability, and interest in performance with the system. Second, implicit data would be collected about the gestures trained, mappings created, and actions during play with the device. This chapter will describe the experimental procedure and results obtained.

4.1 Study Procedure

The study was run in April 2004, at the MIT Media Lab. The subjects were students who responded to an email that sought participants for a study about a "gestural music control" system. Participants were randomly assigned to one of two experimental conditions, which we refer to as A and B. Participants in both conditions began the task with a survey that asked background questions about their age, gender, level of musicality, music performance experience, and music listening habits (see figure E-7 for the text of this survey). Once completed, participants in condition A interacted with the system in pre-configured mode (described below, and hereafter referred to as Presets), after which they filled out a post-part-1 survey (see figure E-8 for the text of this survey). They then progressed to the Training mode, followed by a post-part-2 survey, the text of

which was identical to the post-part-1 survey (again, see figure E-8 for the text of this survey), and then a final survey that asked questions about the experience as a whole (see figure E-9 for the text of this survey). Participants in condition B experienced the same interaction modes, but in the reverse order, starting with the Training mode, and progressing to Presets mode afterwards. In both conditions, data about the triggering of gestures was captured and time-stamped. In addition, the gestures and mapping associations that subjects trained in condition B were saved to disk. The modes and surveys will be commented on further below.

4.1.1 Presets Mode

Subjects in the Presets mode part of the experiment played with the device in a pre-configured state where they did not train new gestures or create manipulation-to-effect mappings. Subjects were given a list of instructions that explained the activity that they would be doing (See figure E-2 for these instructions). They were first shown two videos that explained how the system worked. The first video (See figure E-5 for the video script) was an demonstration of the three gestures that they could use to trigger sounds with the system (including an explanation of proper use of the trigger button), as well as the use of the toggle button to sustain (loop and un-loop) sounds. The second video (See figure E-6 for the video script) was a demonstration of the three input DOF's that were mapped to effect parameters. After viewing the two videos, they were told that they were free to interact with the system for as long as they wanted, but that they would be asked to move to the next part of the experiment when the time came if they were still playing at that point. The subjects were also told that if they felt that they had arrived at a point at which they had exhausted their interest in interacting with the system, they could alert the experimenter and they would be moved on to the next part of the study at that time.

4.1.2 Training Mode

Subjects in the Training mode part of the experiment were able to train new gestures and to create manipulation-to-effect mappings. Subjects were given a list of instructions that explained the activity that they would be doing (See figure E-1 for these instructions).

They were first shown two videos that explained how the system worked. The first video (See figure E-3 for the video script) was an demonstration of how they could explore the available sounds, and train gestures to associate with the sounds (this video included an explanation of proper use of the trigger button), as well as the use of the toggle button to loop and un-loop sounds. The first video also explained how to create manipulation-to-effect mappings with the system. The second video (See figure E-4 for the video script) was a demonstration of the input DOF's that were available for association with effects. After viewing the two videos, they were told to train as many gesture-to-sound associations as they liked, and to make as many manipulation-to-effect mappings as they liked. They were told that they were free to interact with the system for as long as they wanted, but that they would be asked to move to the next part of the experiment when the time came if they were still playing at that point. The subjects were also told that if they felt that they had arrived at a point at which they had exhausted their interest in interacting with the system, that they could alert the experimenter and they would be moved on to the next part of the study.

4.1.3 Data saved to disk

In addition to the surveys that participants filled out during the study, data from their session was captured and written out to a series of log files. In Training mode, each gesture that they trained was saved to a separate text file, and the input-DOF-to-sound mappings were saved to disk as a serialized Java object. In both modes, information about gestures classified and effect modification messages was written to a separate file.

4.2 Study Design

The study involved 25 participants, and was a within-subjects design, meaning that each subject experienced both of the experimental modes. As mentioned above, the Presets mode featured the device pre-loaded with gesture and manipulation mappings, and the Training mode allowed the user to define their own gesture and manipulation mappings. Most of the post-part-1/post-part-2 questions were phrased in a manner like: "How expressive did you feel that you could be in using this system?" and subjects responded

on a 7-point BIDR-style scale that ranged from (1) "not expressive at all" to (7) "extremely expressive". In all graphs, the height of the bar represents the mean of the responses, and the error bars show the standard error.

4.3 Survey-based study Results

This section is broken into three parts. The first part is from the survey after the first mode of the experiment. At this point, subjects in either condition had only experienced a single mode, Presets or Training. The second part is from the survey after the second mode of the experiment, and the third part is from the survey at the end, at which point subjects had experienced both experimental modes.

4.3.1 Gesturing (part 1)

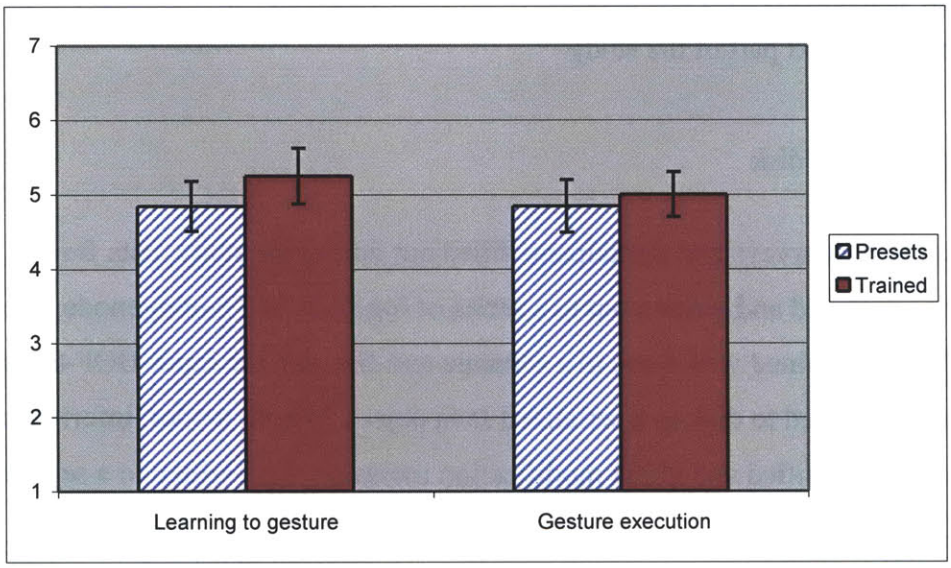

Figure 4-1: Ease of learning (left) and executing (right) gestures (part 1)

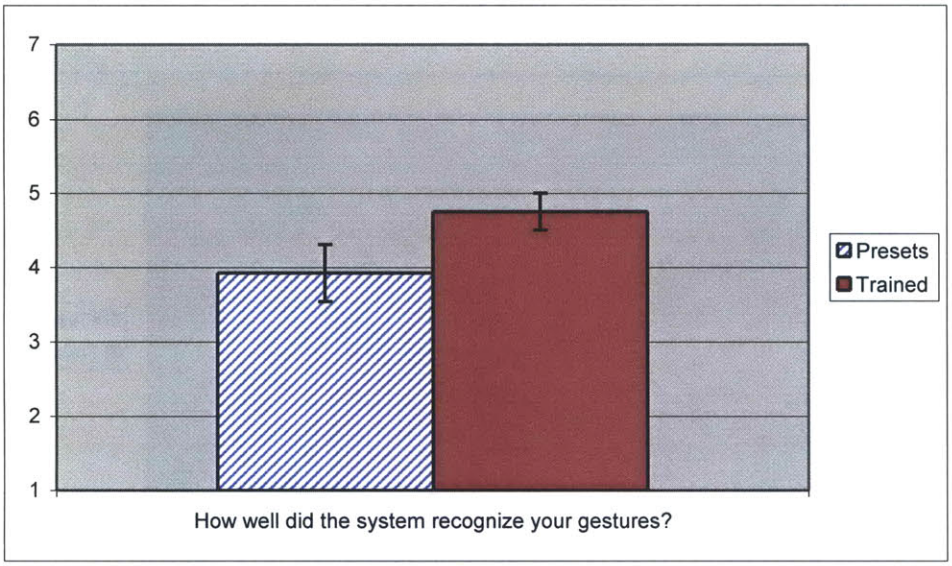

Figure 4-2: Perception of the system's accuracy in gesture recognition (part 1).

When asked "How easy was it for you to *learn* to trigger the sounds that you wanted?" subjects that had just experienced Training mode tended to report that it was easier than those that had experienced Presets mode (avg. 5.25 Training vs. 4.85 Presets). A similar question about creating input-DOF-to-effect mappings showed a less clear preference. When asked "How well did the system recognize your gestures?" subjects in Training mode tended to respond more favorably than those in Presets mode (avg. 4.75 Training vs. 3.92 Presets). These results suggest a more favorable perception towards gesture-recognition in a system that allows custom-trained gestures.

4.3.2 Expressivity and Personalization (part 1)

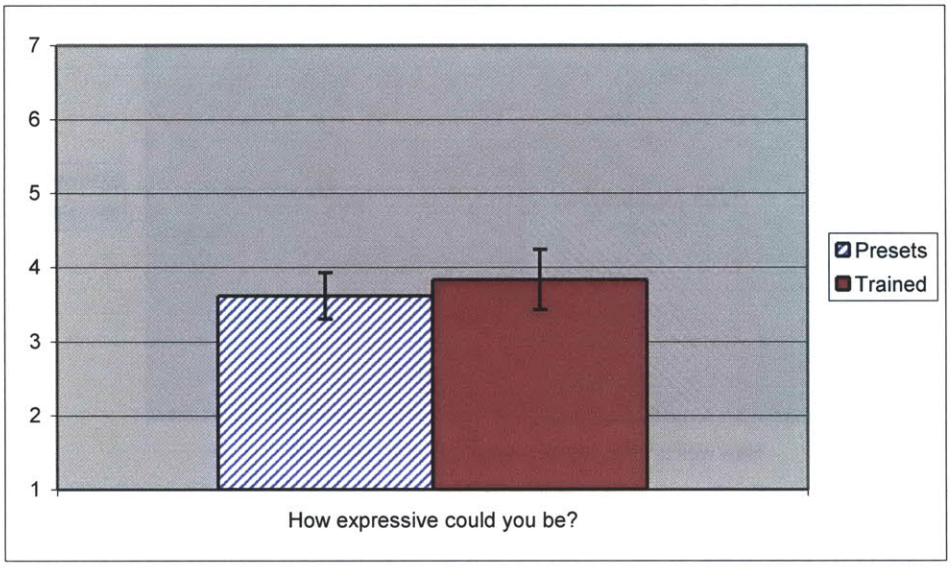

Figure 4-3: Expressivity (part 1)

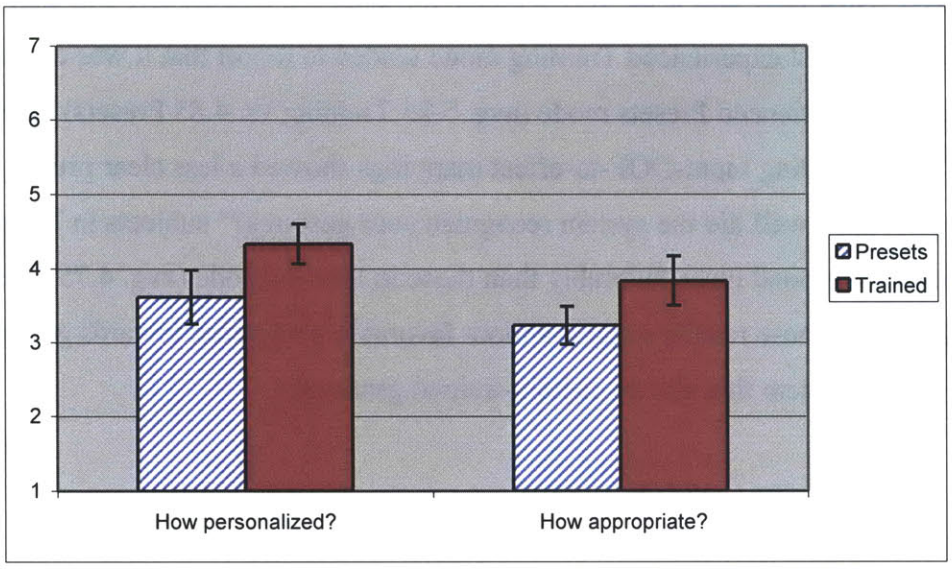

Figure 4-4: Level of personalization (part 1)

Subjects in Training mode rated the system more favorably in terms of expressivity and personalization. When asked "How expressive did you feel that you could be in using this system?" subjects that had just experienced Training mode reported an average of 3.83 vs. 3.61 for those in Presets mode. When asked "How much "personalization" did you feel that this system offered to you?" the numbers were avg. 4.33 (Training) vs. 3.62

(Presets). In response to "Please rate your feelings about the level of personalization you experienced", with the ends of the scale being "far too little personalization" and "far too much personalization" the numbers were 3.83 (Training) vs. 3.23 (Presets).

4.3.3 Enjoyability and future play, performance (part 1)

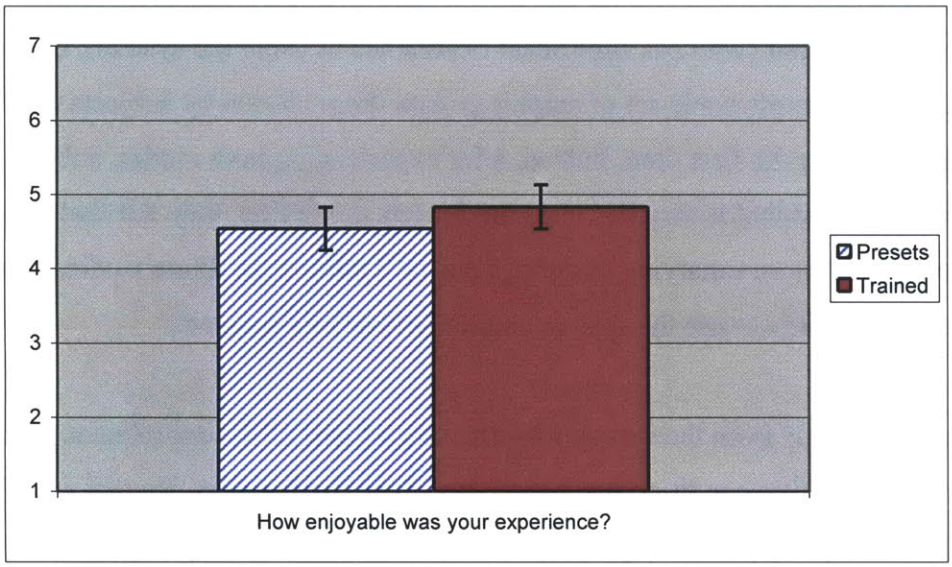

Figure 4-5: Enjoyability (part 1)

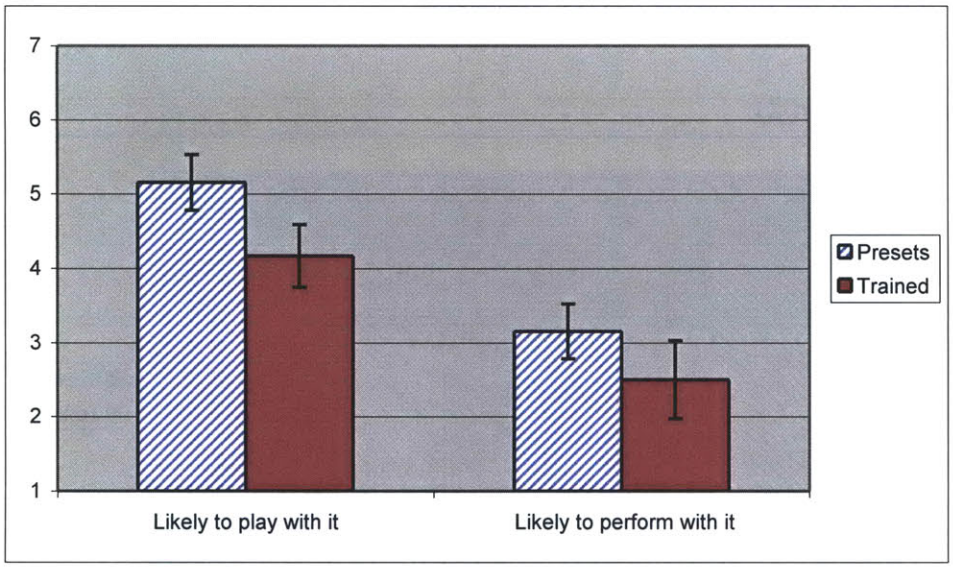

Figure 4-6: Likeliness to be interested in further play, performance. (part 1)

Subjects in the Training mode rated the system as more enjoyable than those in the Presets group (avg. 4.83 Training vs. 4.54 Presets). However, when asked how likely they would be to want to play or perform with the system again, those in the Presets group answered more favorably than those in the Training group (Play: 4.16 Training vs. 5.16 Presets, Perform: 2.5 Training vs. 3.15 Presets). One possible explanation for this is that since the preset gestures and the input-DOF-to-effect mappings were chosen carefully by the author (who has significant experience in using the system), they may have been a more comfortable set of mappings than those chosen by subjects who were using the system for the first time. Indeed, after experiencing both modes, subjects in condition B reported that if they had used the Presets mode first, they felt that they would have been created more satisfying mappings during Training. In future studies, allowing the subjects more time to use the system could alleviate this problem.

The second section is from the survey after the second part of the experiment. At this point, subjects in either condition had now experienced both modes, Presets and Training. Results from this part were mostly consistent with results from the previous part with respect to personalizability, expressivity and enjoyability, but interesting differences found on a few of the questions will be discussed.

4.3.4 Gesturing (part 2)

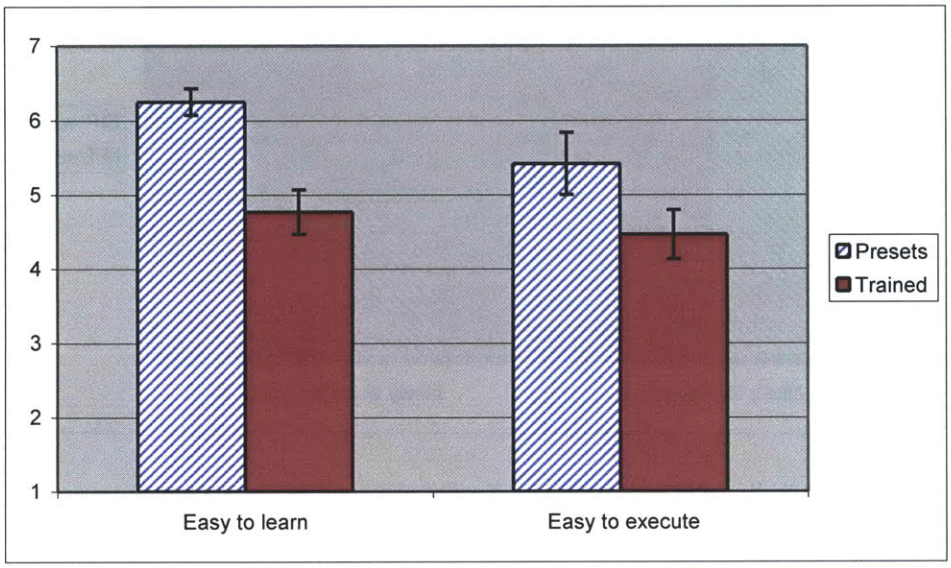

Figure 4-7: Ease of learning (left) and executing (right) gestures (part 2)

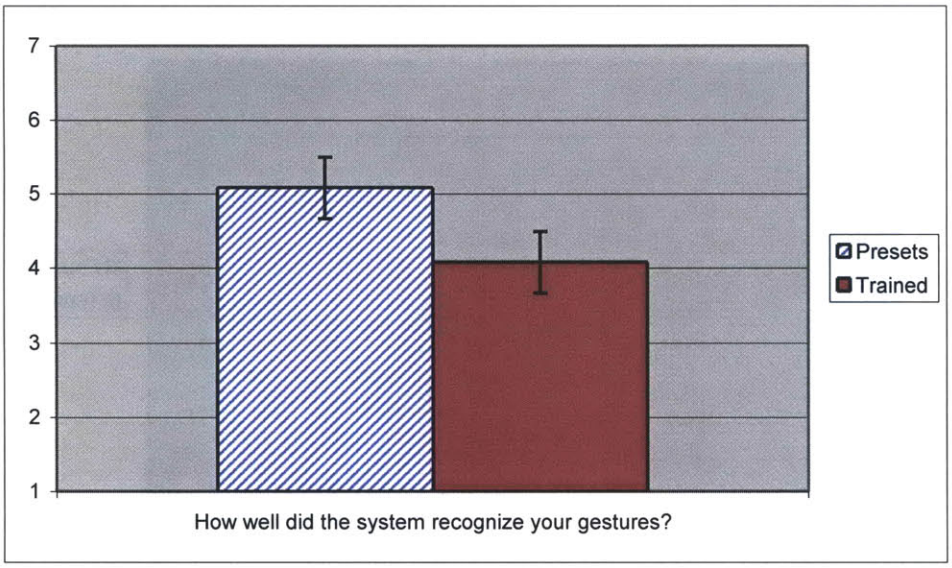

Figure 4-8: Perception of the system's accuracy in gesture recognition (Part 2)

Findings about gesturing after part 2 of the study were inverted from those after part 1. Subjects found the Preset condition easier to learn, and easier to execute gestures within. In addition, perception of the system's accuracy in gesture recognition was higher for subjects in the Presets condition. These are considered to be ordering effects, or it could be that the preset gestures trained by the author were more distinguishable by the system, leading to better recognition rates during Presets mode.

4.3.5 Future play, performance, and novelty (part 2)

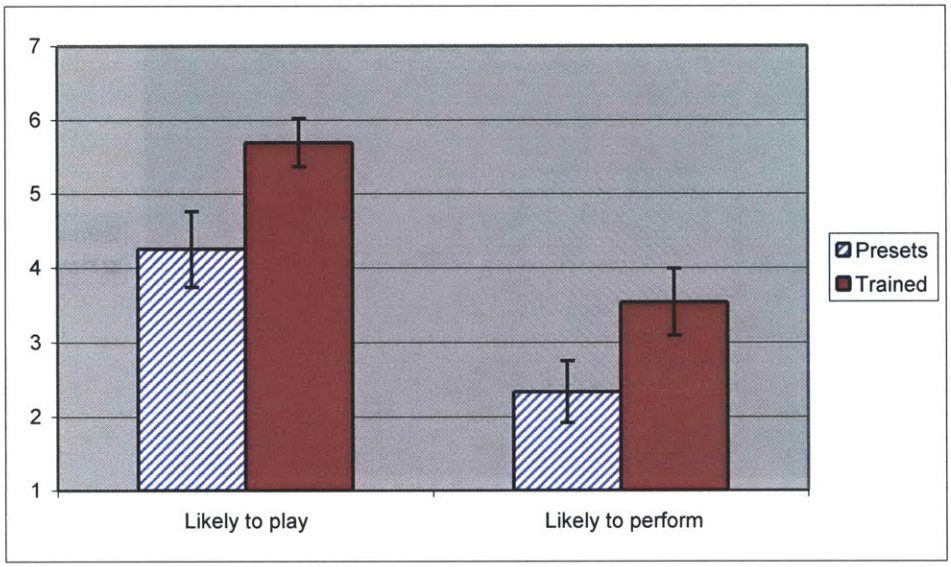

Figure 4-9: Likeliness to be interested in further play, performance. (part 2)

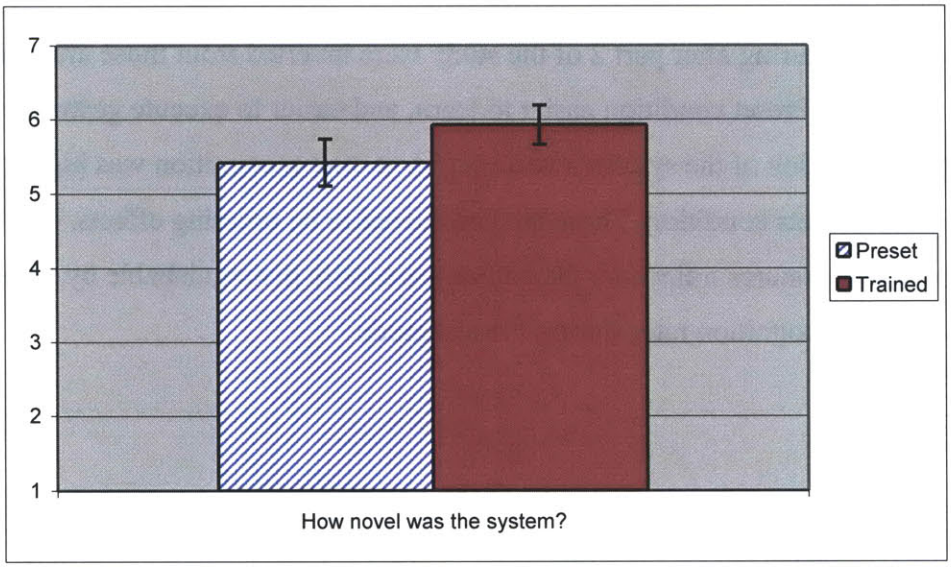

Figure 4-10: How novel was the system? (part 2)

In contrast to their responses after the first part, subjects after the second part claimed to be more likely to want to play (5.69 Training vs. 4.25 Presets) and perform (3.53 Training vs. 2.33 Presets) with the Training mode system than with the Presets. They also tended to rate the Training mode as more novel than the Presets mode (5.92 Training vs. 5.42 Presets). These differences are similar to the "ordering effect" discussed in the

previous section, but here they are thought to reflect an awareness of the differences between presets and open-ended training, and this awareness resulting in a preference for a trainable system.

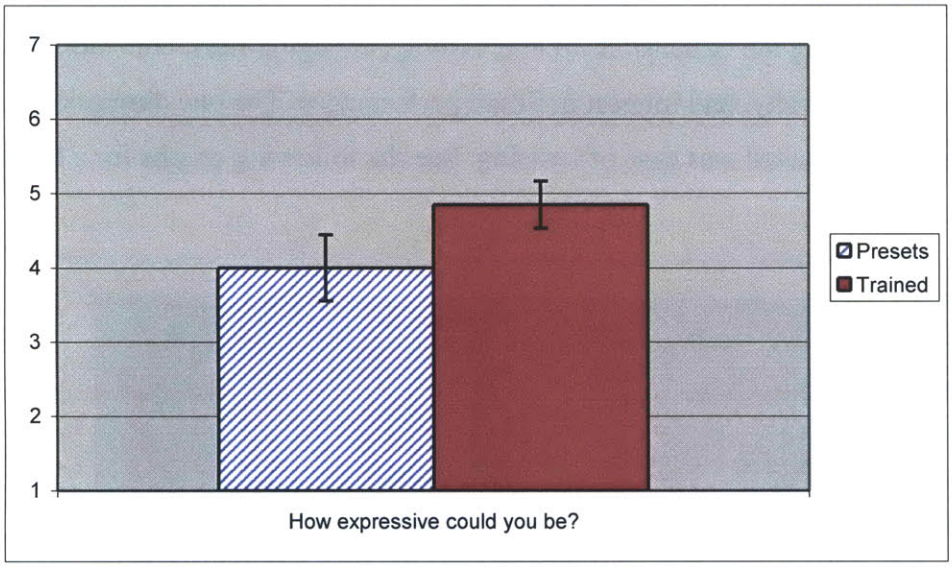

Figure 4-11: How expressive did you feel that you could be in using the system?

After the second part, subjects in the Training condition clearly rated the system more highly in terms of expressivity. This finding is consistent with the questions that asked the subjects about their likeliness to be interested in future play and performance with the device (subjects in Training mode were more likely to be interested in both), indicating a correlation between expressivity and interest in play and performance with a musical instrument. This finding is also interesting when considered alongside the question about ease of gesture learning and execution (in which subjects indicated that the Presets mode was easier). The combination of these two results indicates that subjects would prefer to use the training system, even if it is more difficult to train and execute gestures.

4.3.6 Wrap-up questions (Part 3)

After experiencing both Training and Presets mode, and completing the post-mode surveys for each mode, subjects were asked questions that compared the two systems explicitly. The results of this round of questions indicated a clear preference for the Training mode along the dimensions of expressivity, engaging-ness, enjoyability, personalization, novelty, and interest in future performance. The one dimension that the Presets mode dominated was ease of learning. See the following graphs for a full report.

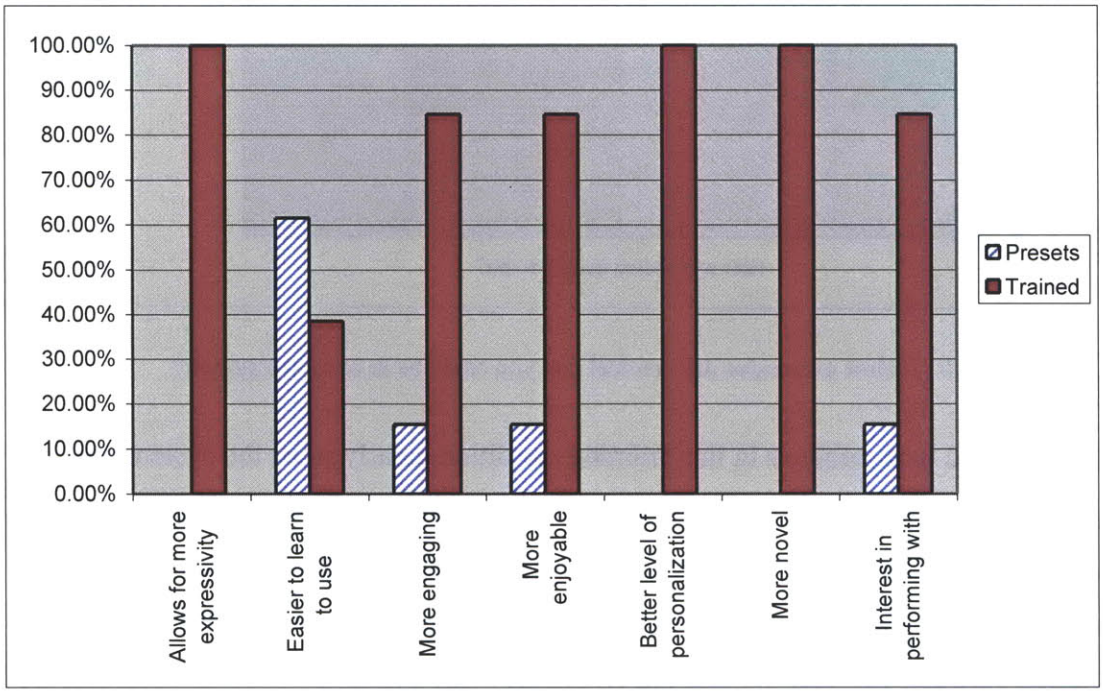

Figure 4-12: Post-study summary questions (group A)

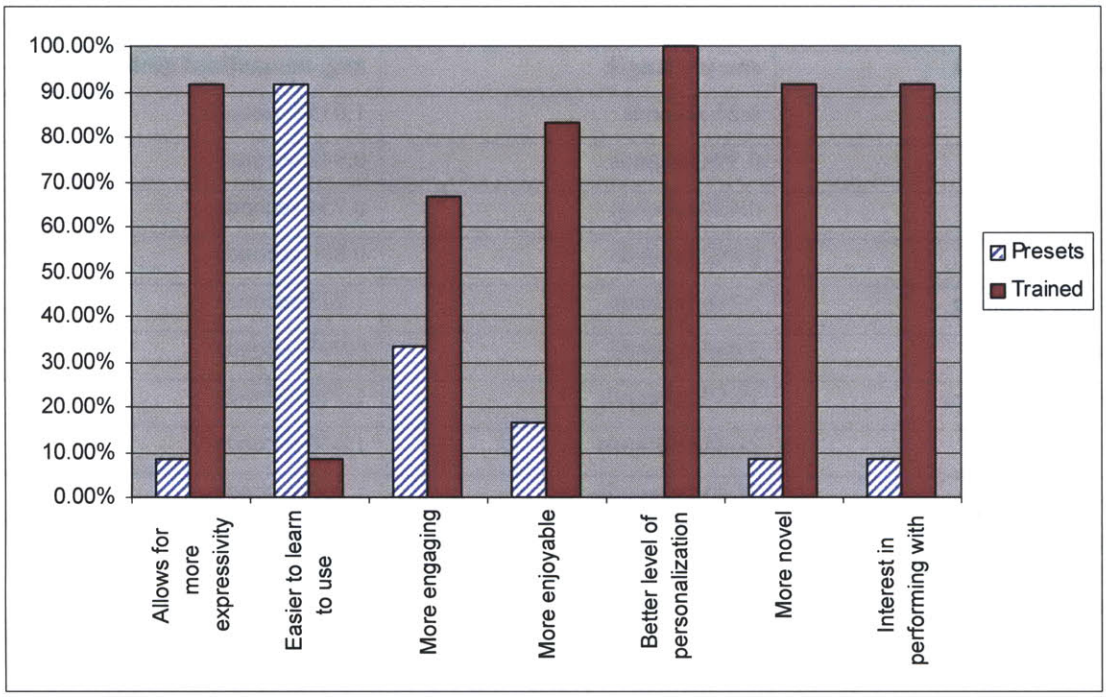

Figure 4-13: Post-study summary questions (group B)

4.4 Results from other data captured

The following section summarizes findings from the data captured during the Training mode sessions. This section is extremely exploratory, as little could be anticipated about the types of gestures that people would assign to the available sounds, and the types of mappings that would be preferred in associating inputs DOF's with effects.

4.4.1 Gesture Length

The first statistic pulled from the implicit data was the length of the trained gestures compared to the length of the triggered sounds. The lengths of the gestures for each subject were first individually normalized by the average gesture length for the subject, to account for differences in gesture-length tendencies between individuals. The results are summarized in the following table.

sound description	sample length	avg. normalized gesture len.
Turntable scratch	0.22 seconds	1.0152 seconds
Tom hit	0.394 seconds	0.948 seconds
Sleigh-bell shake	0.429 seconds	0.754 seconds
Vibraslap	0.862 seconds	0.800 seconds
Rhythmic clapping	2.116 seconds	1.223 seconds
Cymbal crash	2.444 seconds	0.903 seconds
Medium pitch drone	19.597 seconds	1.576 seconds
Low pitch drone	21.358 seconds	1.046 seconds
High pitch drone	21.458 seconds	1.239 seconds
White noise	inf	1.273 seconds

Table 4-1: Sound lengths and average gesture lengths

The following graph plots the same data on a logarithmic scale.

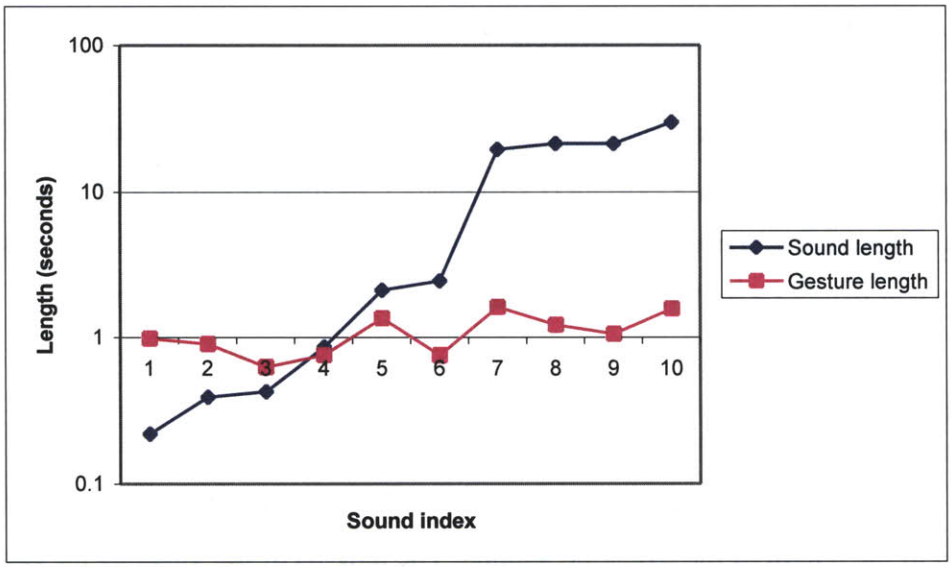

Figure 4-14: Suggesting a correlation between sound length and gesture length

Some of the variation in the data in this representation could be due to noise, but when the lengths of the sounds are be broken into "short" and "long" categories, the trend becomes much more clear. Using 1 second as the dividing line between "short" and "long" sounds, we find that the average gesture length associated with "short" sounds is

0.818 seconds, while the average gesture length associated with "long" sounds is 1.258 seconds. This indicates that the length of the gesture a person will naturally associate with a sound is positively correlated with the length of the sound.

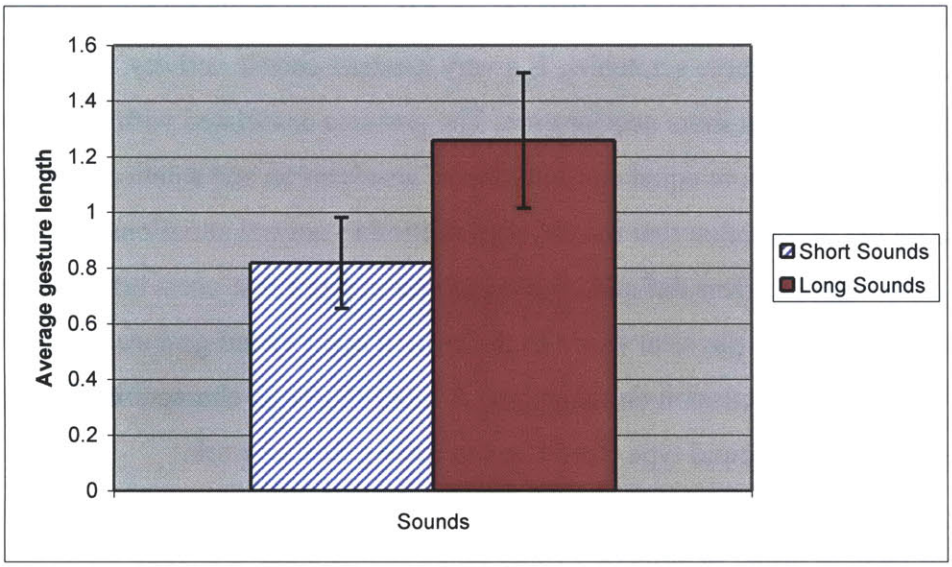

Figure 4-15: Average gesture length for "short" and "long" sounds

4.4.2 Gesture to sound associations

Perhaps more interesting than correlating the lengths of sounds and gestures is a closer look at the features of the gestures being associated with the various sounds. As a reminder, trained gestures consist of data from the accelerometers and gyros – inertial data. The primary measure analyzed here is the per-sensor root-mean-squared (RMS) energy present in the gestures. To preprocess the inertial data, the average energy was computed, per-sensor per-subject, and each gesture's energy profile was then normalized by these values. The following table summarizes the average energy from the accelerometers and gyros from gestures associated with the 10 sounds.

	turntable	tom	sleigh	vibra	clapping	cymbal	drone	drone	drone	noise
acc	16.659	17.053	11.802	12.060	12.027	15.953	10.749	2.612	12.132	2.952
rot	21.079	16.725	13.537	11.635	7.751	14.843	8.369	3.095	10.061	6.691

Table 4-2: Average gestural acceleration and rotational energy per sound

The preceding shows several interesting patterns. First, the "turntable" gestures contain much more energy in rotational than in acceleration. By the same token, the "clapping" gestures contain more energy in acceleration than in rotation. These findings are consistent with the types of motions that are used to produce these sounds under normal musical situations: turntable scratching is a very rotation-centric activity, while a clapping motion features more acceleration. The gestures associated with the "drone" sounds tend to have a more equal combination of acceleration and rotation, which can be explained by the observation that the physical motion in normal situations associated with creating those sounds is less defined. These patterns suggest that users bring experience and metaphors from the physical world to the current open-ended gesture task. (see section 5.1 for more discussion on metaphor) A more thorough characterization of gestural connection to sound type would be interesting further work.

4.4.3 Input DOF to effect mappings

	# subjects who used it	% inverted mapping
Ring modulation	14	71%
Volume	15	33%
Sweeping Band-pass filter	14	36%
Tremolo	19	47%
Pitch-shift	22	91%

Table 4-3: Effects available, usage, and polarity trends observed

The above table has a single sound-modification effect per-row, and it shows two things: The first is the number of subjects (out of 25 total) that created a mapping for the particular effect, which gives an indication of the effect's popularity. Most subjects did listen to *all* of the effects before or during the selection process. The second item shown by the table is the percentage of the subjects that trained an input-DOF for the effect that created an "inverted mapping" for the effect. The polarity of a mapping (inverted vs. non-inverted) refers to how the range of the sensor is mapped to the range of the effect. If the

maximal value from the sensor (for an FSR, given the signal conditioning circuit used, this is when there is the least pressure on it) is associated with the minimal value of the effect, this is called an inverted mapping, and vice versa. The labeling of the polarity ("inverted" vs. "non-inverted") is essentially meaningless – what is interesting however, is the *consistency* of polarity found on certain DOF's. For instance, of the 22 subjects that created a pitch-shift mapping 91% created an inverted mapping. Similar patterns show up in the mapping of ring-modulation, volume, and the band-pass filter. Also interesting is a look at the trends in how people mapped particular input DOF's to effects.

	ring-mod		volume		sweeping filter		tremolo		pitch-shift	
	#	INV	#	INV	#	INV	#	INV	#	INV
top-handle squeeze	2	100%	3	0%	2	0%	5	20%	6	83.3%
carriage twist	6	50%	3	33%	5	20%	3	100%	5	100%
low-handle mid-button	2	50%	4	50%	5	40%	5	60%	6	100%
low-handle low-button	4	100%	5	40%	1	100%	4	50%	5	80%

Table 4-4: Number of mappings created per category, and polarity consistency

Looking at the raw numbers in the chart can be misleading, since effects like tremolo and pitch-shift were more popular in general, thus the high number of mappings from input DOF's to these effects may be part of a distribution. However, the polarity of specific mappings is interesting. For instance 100% of the 6 subjects that made a low-handle-mid-button mapping to the pitch-shift effect created an inverse mapping (meaning that pushing harder on the button caused the pitch to shift upwards). At first this may seem like a natural consequence of the intuition that squeezing harder should cause "more" of the effect, whatever it is. Showing the opposite tendency, however, is the mapping of top-handle-squeeze-to-tremolo. Of the 5 subjects that made that mapping, only one of them trained an inverse mapping.

The patterns collected from the way in which people trained gesture-to-sound and input-DOF-to-effect mappings are interesting in their own right, and could be instructive to future electronic music controllers – especially the majority that design controllers with preset mappings. In future work, the set of input DOF's will be expanded to include a few other affordances, such as tilt and electric field sensing. Using this work as a starting point, a number of more specifically-targeted studies could use the current device or a future version to tease apart an even more detailed look at how people naturally create mappings when presented with an open-ended, trainable controller.

4.5 Study Conclusions

This chapter has presented the design and results of a user study that was run in April 2004 that involved 25 participants. Survey questions interspersed throughout the activity collected the subjective opinions of the subjects. The surveys produced data about engaging-ness, enjoyability, future interest in play and performance, ease of gesturing and novelty. In addition to the surveys, implicit data was collected about the types of gesture-to-sound and input-DOF-to-effect mappings that the subjects created. See the conclusions chapter for more discussion about these findings.

The next chapter will present the conclusions of this work, future directions for work with the system, and some ideas about future applications with the system.

Chapter 5

Conclusions and future work

In this chapter the project is considered in its entirety. We will begin with a summary of this thesis, considering the work to date. Future small-scale improvements to the existing system will be considered, followed by a discussion of long-term applications of an adaptive gestural device.

Overall, the work in this thesis demonstrated that a sensor-rich multi-DOF device can be made to flexibly learn open-ended physical gestures, and can allow a user to create custom assignments of continuous input DOF to output parameters. This device has also been shown to be useful as an investigative tool, helping to understand more about intuitive connections between gesture and sound.

5.1 Summary

The system constructed uses a number of inertial, pressure-based and electric field sensing circuits in order to provide a flexible and adaptive user interface platform. On the hardware end, a microcontroller-based physical device with inertial sensing, electric field sensing, a number of contact-based input degrees of freedom, and serial communication capabilities was constructed. On the PC, software receives data from the device, storing and interpreting it. When configured in training mode, the routines allow custom inertial

gestures to be trained and associated with output sounds, and permit continuous input degrees of freedom to be flexibly assigned to sound modification (effect) parameters. During play mode, the software on the PC uses a dynamic time warping algorithm to classify incoming gestures based on the user's trained class models, and continuously updates sound modification parameters based on the state of the continuous input DOF's. The most recent sound triggered can be put into a sustained (looping) state during play mode by pressing up on a toggle button. Looping sounds can be layered, and the most recent looping is canceled by pressing down on the toggle button. Pressing the toggle button directly in cancels all looping.

A user study was designed and run, in which subjective impressions of the system, along with implicit performance and configuration choices were collected. Data from a survey showed that users found the trainable system more engaging and expressive. Observation of subjects during the study indicated a high sense of engagement during the training phase. There was inevitably a visible "a-ha" moment when the subject first trained a custom gesture and tested it, triggering the associated sound. At that moment, subjects would typically sit up straighter, move closer to the computer, and some even made utterances of pleased surprise like "cool!" Similar excitement was observed at the moment with an input DOF became associated with an effect, and the mapping suddenly became enabled. After experiencing both modalities (Training and Presets), subjects using the Training mode reported being more likely to want to play and perform with the system in the future, and that they found the system more expressive. In addition, this group also rated the system favorably in terms of its amount and appropriateness of personalization. A survey at the end of the study showed a clear preference for the training-mode system in the areas of expressivity, engaging-ness, enjoyability, personalization, novelty, and future interest in performance. The only area in the final survey in which the Presets mode was rated more highly was in ease of learning to use the system. This problem would likely be mitigated in the future by giving players more time to familiarize themselves with the device (the entire experiment, including the Training and Presets mode lasted less than 1 hour). Overall, subjects found the system very compelling, and during the study several of them pushed the system into some sonic

and rhythmic spaces that were fascinating to the author and quite different than anything he had previously created with the device.

Mapping data captured during the user study suggests that metaphors from experience making sounds in the physical world may be an underlying force in the assignment of custom gestures. For instance, the gestures that subjects associated with a turntable "scratch" sound featured more activity on the gyros than on the accelerometers This pattern indicates a tendency towards gestures with rotational motion, which is consistent with the way in which an actual turntable "scratch" is executed. By the same token, gestures trained for a percussive drum-hit sound tended to contain more acceleration energy than rotation, which is consistent with the physical action of striking a drum. The explicit use of interface metaphor is an established human-computer-interaction (HCI) principle in which a designer couches an interaction or representation in a framework that is conceptually familiar to the user [Carroll, et al. 1988]. The HCI concept of metaphor can be related to the current data in the following way: The FlexiGesture is open-ended enough that subjects may be using metaphor intuitively, their gestures related to actions that they know from a lifetime of experience with sounds in the world, even though no metaphor is explicitly designed into the system. This spontaneous use of metaphor parallels the overarching FlexiGesture philosophy– rather than trying to build in the "right" affordances (or metaphors, in this case), the system is flexible enough that users can train it in a way that makes the most sense to them individually. Finally, gesture-length was shown to be correlated with sound length, and clear preferences were found for the assignment of input DOF-to-effect mappings. These findings from the implicit data are interesting, and they suggest an exciting future use for this device as a refined "musical affordance laboratory" tool, enabling the systematic discovery of intuitive mappings for computer-generated sound and other media.

5.2 Future Work

As the intent of this thesis was to provide a proof-of-concept, working system that can flexibly learn gesture-to-sound and continuous-manipulation-to-effect associations, there

are a number of improvements to the hardware and software that could be made that bear further discussion.

A software-related feature that was considered but not included in the final version is a tilt-based navigation of the sound space, rather than the current list traversal. Coupled with a synthesizer that had continuously variable parameters, tilting could provide an interesting 2-dimensional navigation metaphor. An idea that was considered early in the design process would use a genetic algorithm/hill-climbing metaphor in starting at a "seed" set of synthesizer parameters, with a neighborhood of mutations from the seed accessible in various directions by tilting the device off-axis around in a circle. When a neighbor was found that the user considered more favorable than the seed, a button press could make that neighbor into the new seed, which would generate a correspondingly new neighborhood of mutations. A related tilt-based navigation interface for physical space is discussed in [Eslambolchilar, et al. 2004].

A hardware feature that could aid in the aforementioned tilt-related navigation would be a microcode tilt-sensing algorithm that controls the individual LED's around the edge of the circular PCB. The accelerometers on the inertial layer can be used to sense the tilt of the device by calibrating to a baseline of gravity, and detecting slow-moving changes to the baseline. The circular PCB was built initially to support rich visual feedback, and each of the 64 LED's can be individually turned on and off, which could enable a "water-in-a-pan" style visualization of the device's tilt. The LED's could also be used in many other ways to give the user and audience relevant feedback about the internal state of the device, perhaps lending a greater sense of transparency and causality to a performance.

A few interesting variants and uses of the dynamic time-warping algorithm could be tried as well. In computing model gestures, any number of in-class examples could be combined by running the DTW algorithm on pair-wise on each pair of gestures, then warping the N-1 gestures to the 1 that had the lowest error to the others. Once warped to the best-fit gesture, the average of all N example gestures could be taken as the class

model. As suggested in chapter 3, classification speed could be improved in the future by enabling optimization and using just-in-time compilation of the Java bytecode.

A variant on the DTW method used in this work could be to adapt the features employed in each run of the algorithm based on which model is being tested against. Each of the models could potentially have a different sensor importance profile, which would be the subset of sensors that were an important part of that particular gesture. When testing against a model gesture, a modified DTW could be run that only analyzed the relevant sensor data streams, given the model's profile. This could potentially result in more accurate classification. In addition, a graceful-recovery feature could be added that would trigger some "default" or more roughly-classified sound in the cases that classification fails. These would remedy the occasional situation in which no sound is produced in response to an executed gesture that has high error compared to all model gestures.

As an alternative to dynamic time warping, the continuous hidden markov model algorithm could be applied to the gesture classification task in the system. In contrast to the discrete HMM formulation, the continuous HMM does not require the data to be drawn from a symbolic alphabet. A continuous HMM formulation would also allow for continuous estimation of gesture class as the data entered the system and was processed in real-time. This configuration could not only speed up classification, but could allow for some form of sound output to be produced before the gesture was fully completed – a feature that some users suggested as a possible improvement. The tradeoff in using a HMM representation as opposed to the current DTW implementation would be that the model could require more training data before becoming a useful classifier.

Finally, as mentioned earlier, the continuous input DOF data could be fed into the gesture training and classification routines along with the inertial data streams. In addition, tilt and electric field sensing inputs could be added to the set of inputs to the system.

5.3 Future Applications

As a music controller, the FlexiGesture has already broken new ground in being a novel physical device built specifically to support flexible and adaptive behavior. One direction of future applications for the system could be the control of other rich media during performance, such as mixing video or dramatic control of theatrical lighting and effects. In addition, since the novelty of the system is in its quick and responsive adaptation, the training of the gestures and manipulations could be an integral part of a performance, introducing the audience to the concept of a real improvised dialogue between performer and device.

Moving past the idea of a music controller, future applications of an adaptive system like this extend into a wide range of hand-held devices. A living room "wand" that could learn to trigger the opening of windows gestured at, or dim the lights based on a squeeze or tilt could be compelling and flexible tool. A television remote-control with inertial sensing inside could be taught to associate gestures with different television channels, providing random-access to favorite stations without having to squander surface real-estate with a large number of single-purpose buttons.

Computer gaming is another application area that the device could be well suited to. Novel commercial and experimental game controllers push the edge of input devices, many of them featuring a multi-degree-of-freedom set of input affordances, typically in the form of digital buttons for fingers and thumbs, or two-handed twisting of the controller [ActiveWindows website]. Some controllers even sense tilt as an input [Saitek P2000 Manual] [Joystick Review website]. Equipped with a joystick-style interface (USB) to the computer, the device could be used for game control, or for any number of media-centric applications for which game controllers have been appropriated [Manor] [Sarlo].

There are other objects that could be fitted with the requisite sensing and processing capabilities in order to emulate the behavior that the FlexiGesture is capable of. For instance, the ubiquitous cellular phone, but equipped with accelerometers inside (phones with this feature will be on the market soon), could become an adaptive musical instrument, enabling impromptu "jam sessions" on the sidewalk or subway. Based on a common device like a phone, a system like the FlexiGesture could also allow people to share gestural music associations, beaming them from device to device with RF or infrared communication. Implemented on such a widespread platform, the system could aggregate large-scale information about gesture-to-sound associations and interface preferences.

5.4 So, will it replace the electric guitar?

There is no short answer to this question. In its current form the FlexiGesture can't beat a piano or a horn at playing a familiar melody, and the guitar has an incredible amount of "rock-star-cool" momentum that makes it a juggernaut in the world of musical instruments. However, the FlexiGesture illustrates a fresh paradigm. It is the first of a new class of instruments that reject the "one-size-fits-all" model in which the player practices for years to adapt his gestures to suit the instrument, or struggles with a poorly designed user interface to change presets to his liking. An instrument that learns the player promises a more satisfying and expressive musical experience, and opens up a user base that is far larger than the current number of musicians. Upon hearing about the device, numerous non-musicians replied "that's just what I need!" recounting stories of frustration with existing instruments, but an underlying desire to play music. These accounts suggest that an adaptive musical instrument could provide a compelling musical interface to people who never found success with a traditional instrument. In addition, the increased sense of personalization that users felt in training mode indicates that a greater sense of attachment could be developed for an adaptive musical instrument. For existing electronic musicians, the device would help solve the "laptop" problem by giving the audience a better sense of musical causality in connecting the performer's physical actions to the sounds being produced. And to the guitar players, it offers a new expressive

set of affordances, and capabilities that are go beyond, or at least in a different direction, than what can be done with a guitar.

To put forth a concrete example of the FlexiGesture in use, an ensemble of players with new musical instruments was created at MIT in March 2004, as part of a visiting residency by experimental composer and musician John Zorn. In the context of a rule-governed improvisation session, the FlexiGesture was played by the author, and was an expressive and contributing member to the overall piece.

Here's what some great musical thinkers of our time have said about the FlexiGesture:

David, man, that's really cool!

- John Zorn, experimental composer and musician (during his artist residency at MIT in March 2004)

It's very inspiring. Very inspiring. This thing has a great future. Very nice.

- Tan Dun, Grammy and Academy award-winning composer (at the Media Lab in April 2004)

Appendix A

Abbreviations and symbols

API	Application Programming Interface
DLL	Dynamically Linked Library
DOF	Degree Of Freedom
DTW	Dynamic Time-Warping
EFS	Electric Field Sensing
FSR	Force Sensitive Resistor
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HMM	Hidden Markov Model
IC	Integrated Circuit
JIT	Just-In-Time (compilation)
JNI	Java Native Interface
MIDI	Musical Instrument Device Interface
OSC	Open Sound Control
PCB	Printed Circuit Board
PC	Personal Computer
PD	Pure Data
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

Appendix B

Schematics and PCB layouts

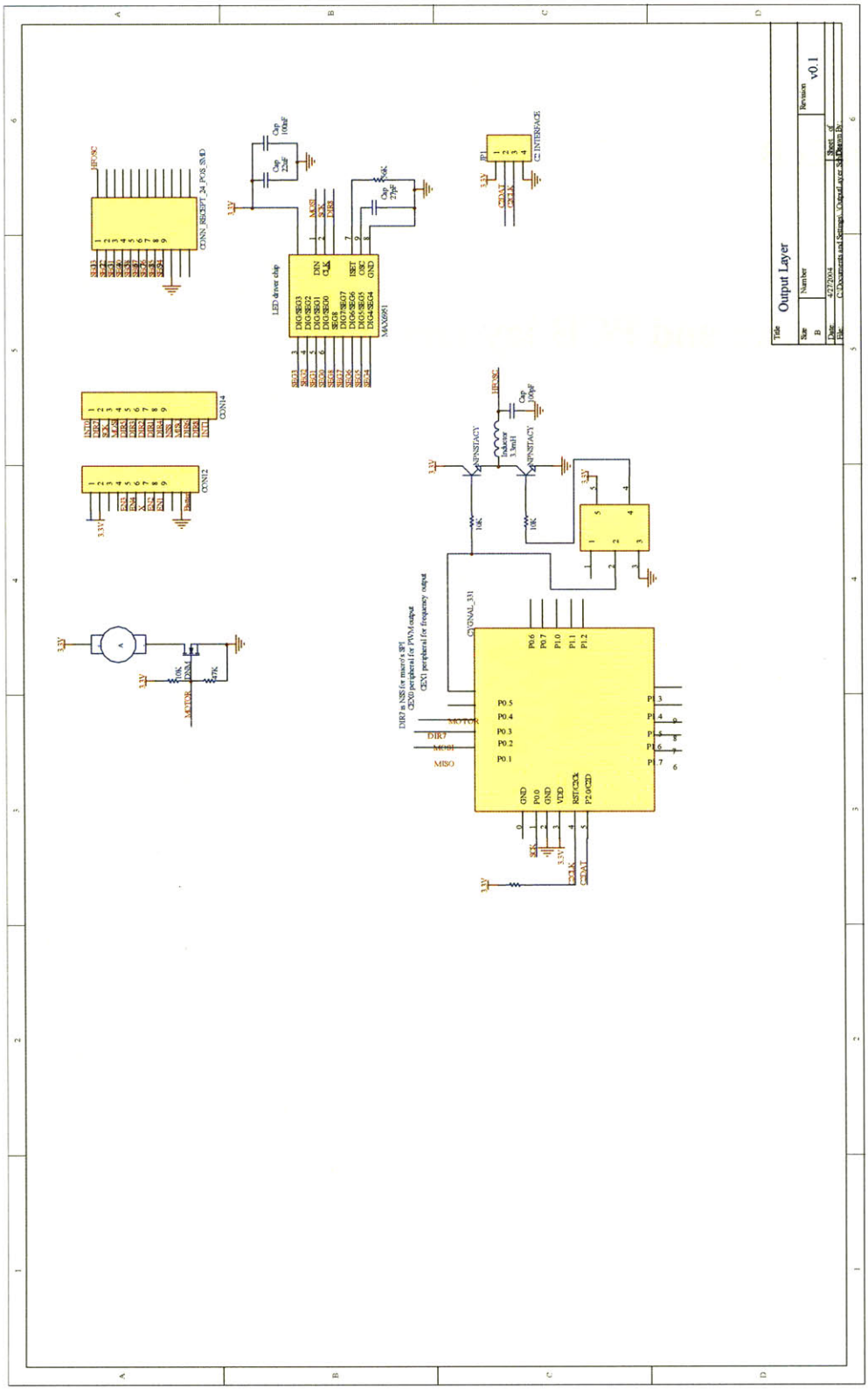

Figure B-1: Schematic for the Output Layer

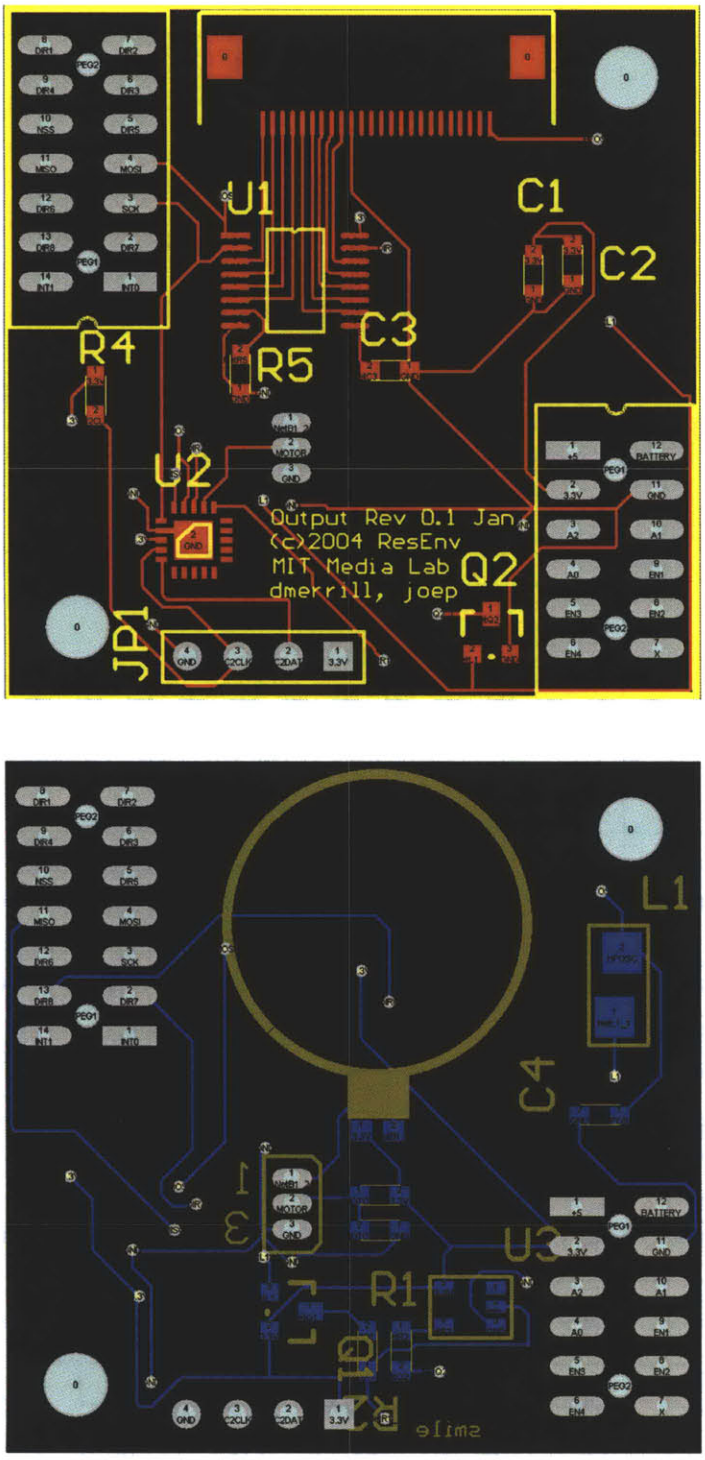

Figure B-2: PCB for Output Layer top (top) and bottom (bottom)

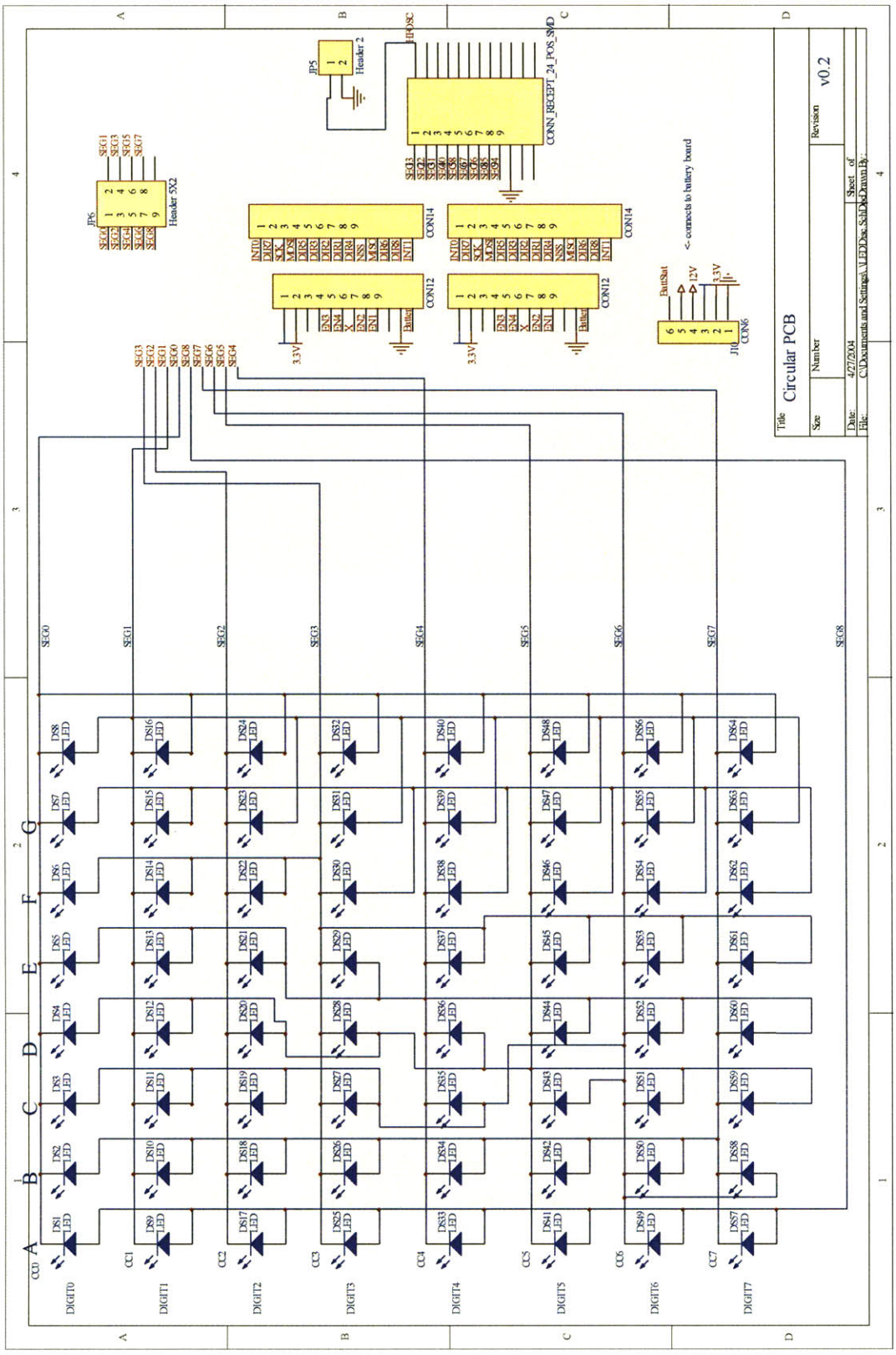

Figure B-3: Schematic for the Circular PCB

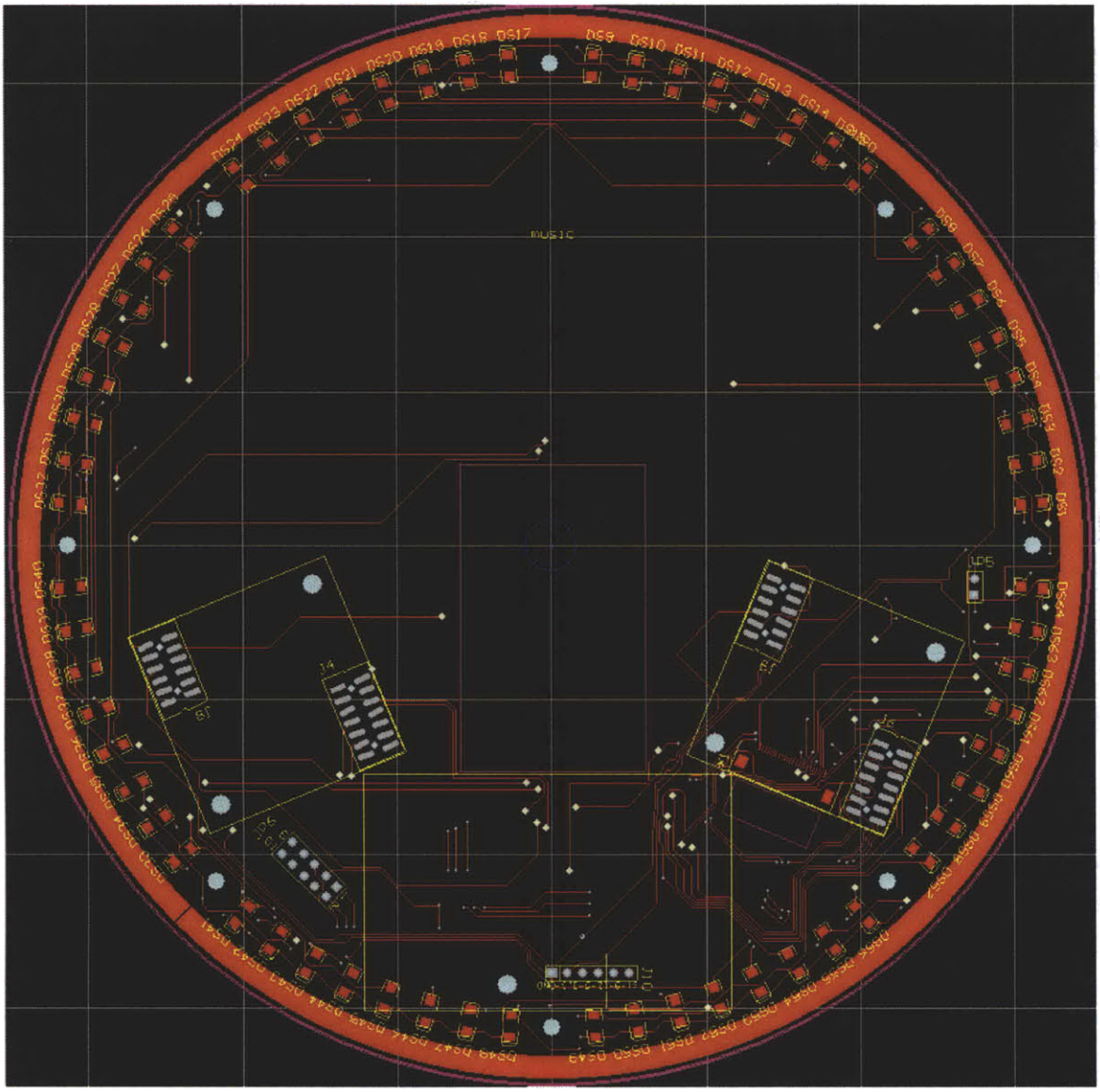

Figure B-4: PCB for Circular PCB (top)

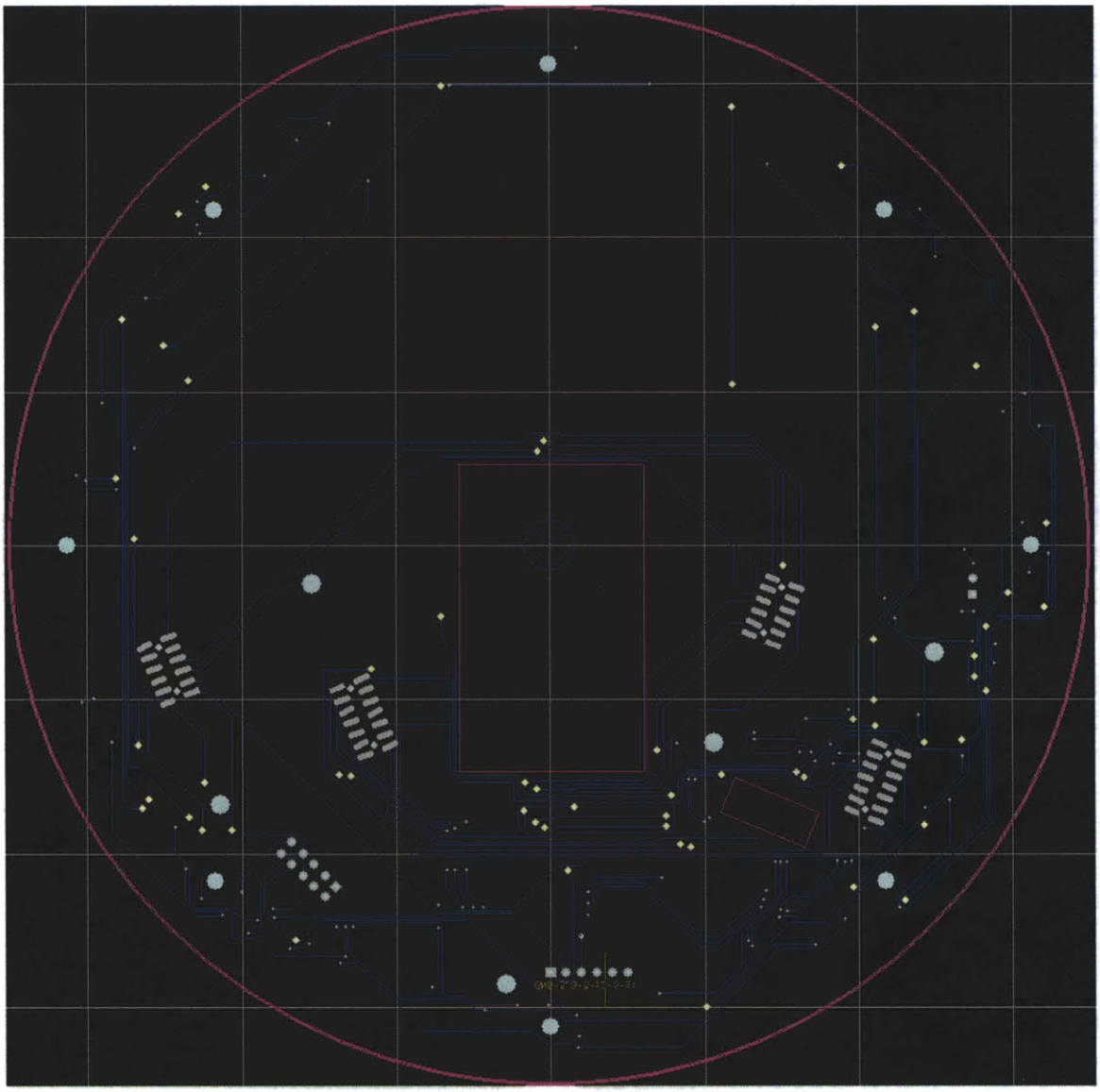

Figure B-5: PCB for Circular PCB (bottom)

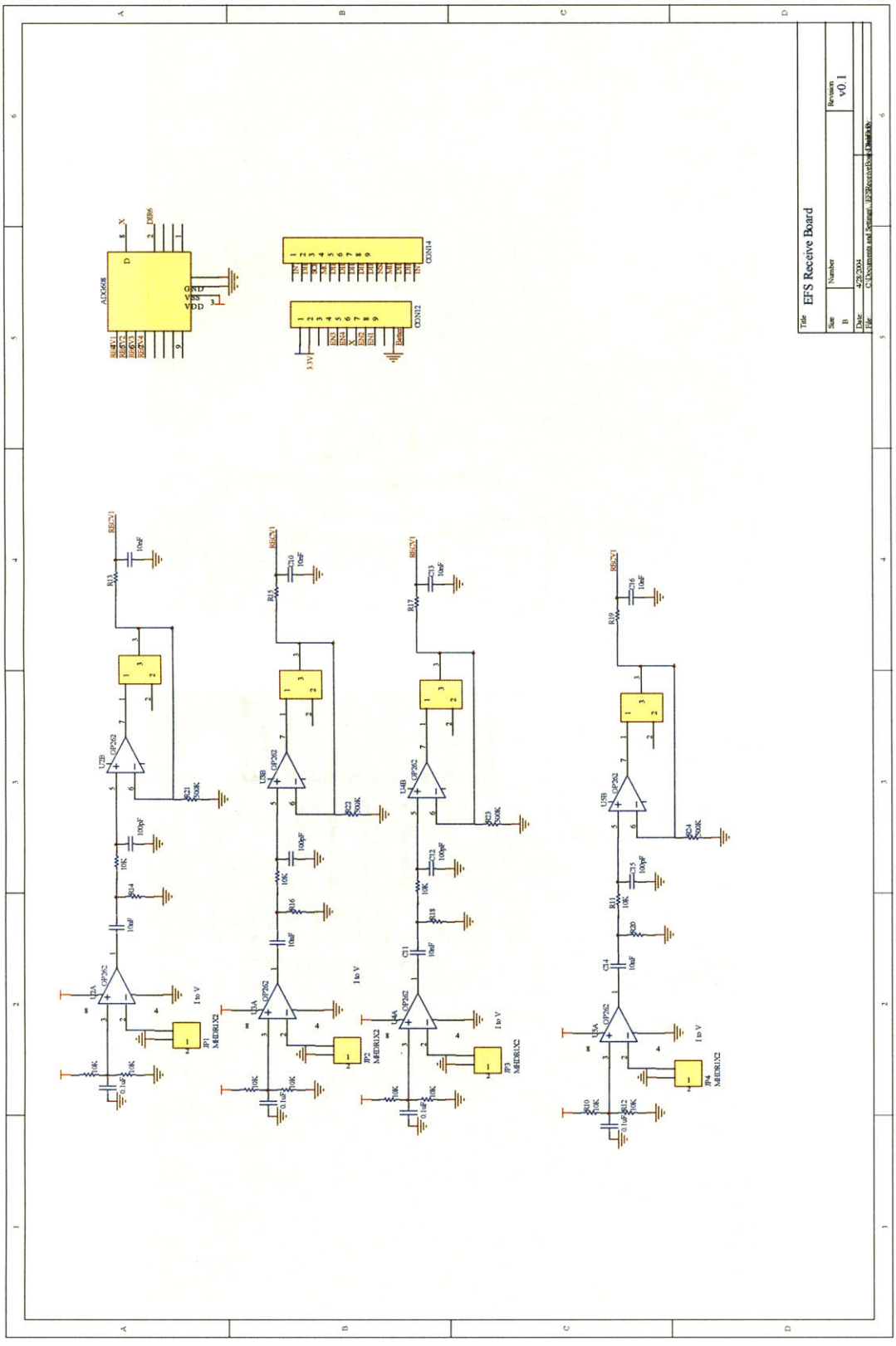

Figure B-6: Schematic for the Electric Field Sensing Layer

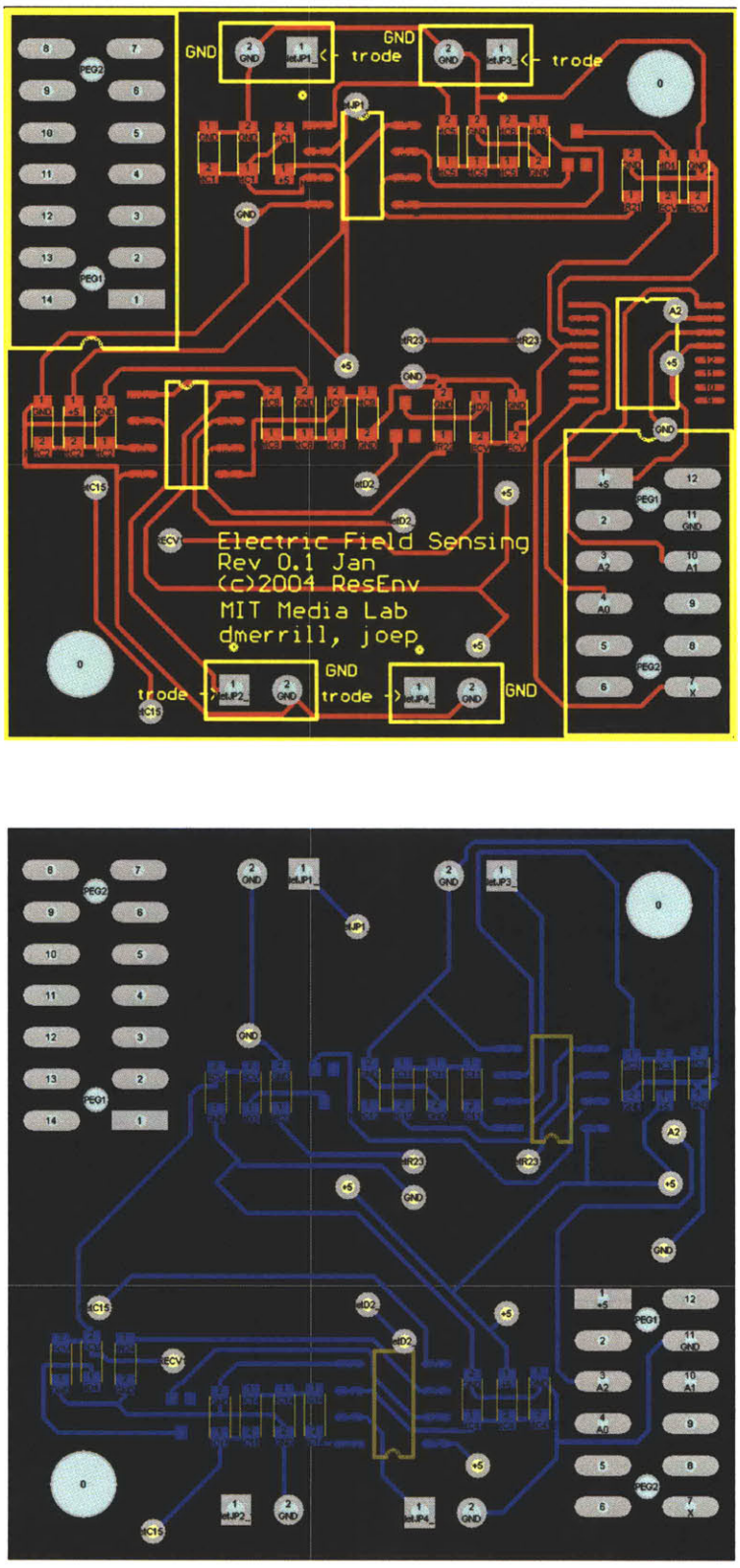

Figure B-7: PCB for the Electric Field Sensing Layer top (top), and bottom (bottom)

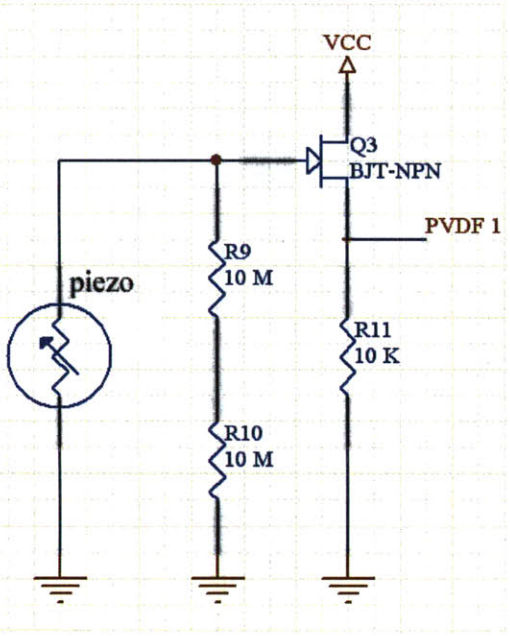

Figure B-8: A piezo signal-conditioning circuit, before re-purposing for FSR use.

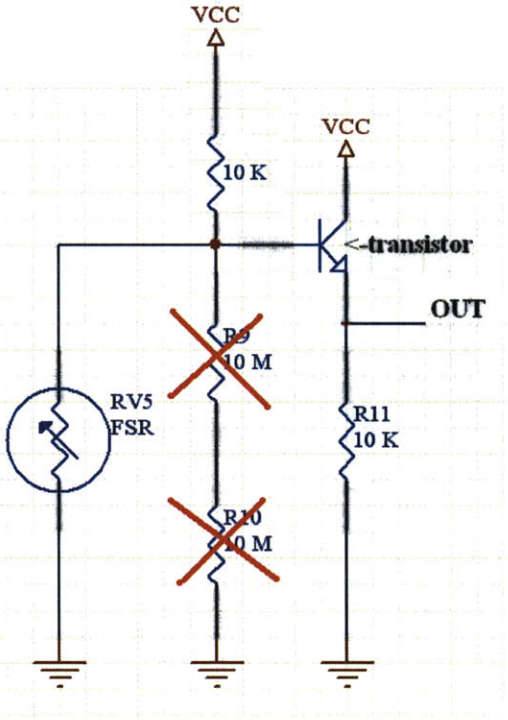

Figure B-9: The piezo signal-conditioning circuit, after re-purposing for FSR use.

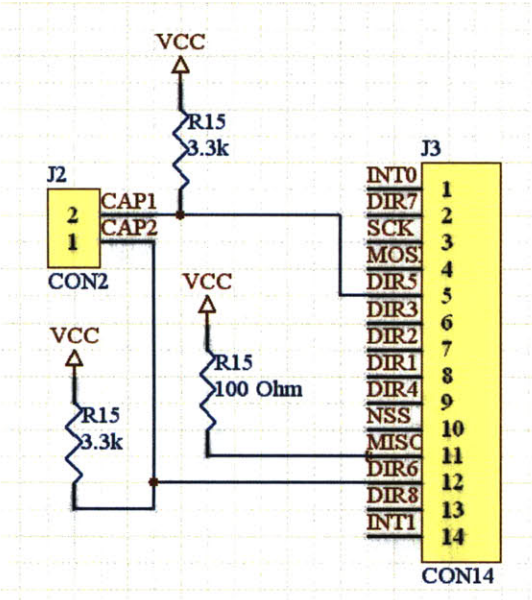

Figure B-10: Connection diagram for the 2 digital buttons

The header connections for 2 primary capacitive sensors on the tactile board were re-appropriated to carry signals from the two digital buttons. A pull-up resistor was connected to each, and they were sent back to the output layer on lines DIR5 and DIR6. The header also provides a GND connection for the buttons.

Appendix C

Embedded C code

```
/* thesis.c
 * --------
 * Author:   David Merrill
 * Modified: February 2004
 * Borrows from code written by Stacy Morris, Ari Benbasat
 */

#include <stdio.h>
#include "206.h"
#include "output_leds.h"
#include "thesis.h"

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void init(void);
void collectAnalog(void);
void collectAccels(void);
void transmitData(void);
void trans3Nibbles(unsigned char,unsigned char);
void setMUX(unsigned char in);
void transmitFSRData(void);
void UART0_Init (void);
void UART0_ISR (void);
void send_uart_byte (unsigned char byte);
unsigned char check_uart_tx_buf (void);
unsigned char check_uart_rx_buf (void);

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define SET_REC   CTR0=1; TEnable=0;
#define SET_TRANS CTR0=0; TEnable=1;
#define BUFFER_SIZE      32

// #define DO_BIDIRECTIONAL
#define DO_LEDS
#define DO_DATACOLLECT

#define LOW_PRESSURE_THRESHOLD 2000
#define HIGH_PRESSURE_THRESHOLD 2500

// Declare ports
sbit CTR1               = P3^1;
sbit CTR0               = P3^0;
sbit TEnable    = P2^7;
sbit SPWRESET   = P2^5;

// Declare direct lines
sbit DIR1               = P2^4;
sbit DIR2               = P2^5;
sbit DIR3               = P2^6;
sbit DIR4               = P3^7;
sbit DIR5           = P3^6;
sbit DIR6               = P3^2;

// Plug in lookup table
idata unsigned char lookup[64] =
{23,27,29,39,43,45,46,51,53,54,57,
58,71,75,77,78,83,86,89,90,92,99,
101,102,105,106,108,113,114,116,
135,139,141,142,147,149,150,153,
154,156,163,165,166,169,170,172,
177,178,180,184,195,197,198,201,202,
204,209,210,212,216,225,226,228,232};
```

```
#define MUXPORT                 1
#define MUXPIN                  7
// Mux stuff
sbit X  = P1^7;
sbit A0 = P1^2;
sbit A1 = P1^1;
sbit A2 = P1^0;
sbit EN1= P1^3;
sbit EN2= P1^4;
sbit EN3= P1^5;
sbit EN4= P1^6;

sbit DATA = P2^0;
sbit ASK = P2^1;
sbit HAVE = P2^2;

//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------
lu gyrox;
lu gyroy;
lu gyroz;
lu accelx;
lu accely;
lu accelz1;
lu accelz2;
unsigned char idata rx_uart_buf[BUFFER_SIZE];
// unsigned char idata tx_uart_buf[BUFFER_SIZE];
unsigned char rx_uart_head,rx_uart_tail;
unsigned char status;
unsigned char transmit_data;

bit in_cycle,pause,dirty;
lu bend1,bend2,fsr1,fsr2,fsr3,fsr4,fsr5,fsr6, dir5_switch, dir6_switch;

lu capA, capB, cap2, cap3;

lu timestamp,checkedBit;

unsigned char milLow, milHigh;
unsigned char counter;

int curr_pressure_threshold = 2000;

void milliClock() interrupt 5
{
}

/* Set up the oscillator */
void oscinit(void)
{
        int delay;

        OSCXCN = 0x66;          // Enable external crystal
        WDTCN = 0xDE;           // disable watchdog timer
        WDTCN = 0xAD;

        delay=256;                      // Delay >1 ms  before polling XTLVLD.
        while(delay--);

        while (!(OSCXCN & 0x80));       // Wait until external crystal has
                                                                // started.
        OSCICN = 0x0C;                          // Switch to external oscillator
        OSCICN = 0x88;                          // Disable internal oscillator; enable
                                                                // missing clock detector.
```

Figure C-1: thesis.c embedded C code

```
        while (!(OSCXCN & 0x80));       // Wait until external crystal has
                                                                    // started.
        OSCICN = 0x08;                          // Switch to external oscillator
}

void UART0_Init (void) {
        rx_uart_head = 0;
        rx_uart_tail = 0;

        // Setup serial comm (using timer 1)
        PRT0MX |= UARTENABLE;                   // Turn on UART (0x01)
        CKCON |= T1RAM;                                 // Use SYSCLK, instead of SYSCLK/12
        SCON = UART8BIT | RECEIVEON;    // Setup serial port
        TMOD = T1TIMER | T16BITRELOAD;  // Setup up serial timer (Mode 2: 8-bit counter/timer with auto-reload)

        // set TL1 = 0x00, TH1 = 0xFA
        SET_TIMER(1,0xFA,0x00);         // FA -> 115200, F4 -> 57600 Baud timeout for 22.1184 MHz crystal
        START_TIMER(1);
}

unsigned char check_uart_rx_buf (void)
{
        if (rx_uart_head - rx_uart_tail == 1) {
                return (254);
        } else if ((rx_uart_head == 0) && (rx_uart_tail == (BUFFER_SIZE-1))) {
                return (254);
        } else {
                return (0);
        }
}


void send_uart_byte (unsigned char byte)
{
}

// Setup function
void init(void)
{
        oscinit();
        UART0_Init();

        ASK = 0;

        // Setup ADC
        ADCEN = 1;                      //Turn on ADC
        AMX0SL = 0x20;          //Turn on the analog mux
        ADCTM = 1;                      //Turn on tracking mode
        ADC0CF = SAR16|G1;      //Set the ADC clock and prescaler
        REF0CN = VDD;           //Internal reference

        //Setup accelerometer timer 0
        TMOD |= T0TIMER | T016BIT;

        // more stuff pertaining to the serial here..
        EIE1 |= 0x01;                   // EXTERNAL INTERRUPT ENABLE 1
    EA=1;                                       // Make sure global interrupts are on
        IE |= 0x10;                             // turn on the serial port interrupt
        RI=0; TI = 0;                   // Ready to receive and transmit

        // Inputs should be open drain (0) and high impedance (this is the default
        // Outputs should be push-pull (1) and default low
        P0TOCF = 0x01; // (P0.0 is TX, and P0.1 is RX)
        P0T1CF = 0x7F;  EN1=EN2=EN3=EN4=0;
        //P0T3CF = 0x67;
        P0T3CF = 0x23; // want to make sure P0.5 and P0.6 are open-drain

        // set up Port 2 (the SPI)
        P2T2MX =  0x01;     // set the SPI enable bit, enable weak pull-ups for all ports
        P2T2CF =  0x0D;         // P2^0, P2^2, P2^3 should be push-pull (for SPI) [0: SCK, 1:
        P2MODE =  0xFF;         // setup all of P2 to be digital I/O (instead of analog input

#ifdef DO_LEDS
// set up the SPI-specific registers

/* how I set up SPI0CFG (bits set)
 * bit 7 -> 0 : CKPHA (spi clock phase) ; data sampled on first edge of SCK period
 * bit 6 -> 0 : CKPOL (spi clock polarity) ; SCK line low in idle state
 * bit 5/4/3 -> 0 : which of the up to 8 bits of the SPI word have been transmitted
 * bit 2/1/0 -> 1 : number of bits to shift in/out of the SPI register during master-mode
 *                  111 means that 8 bits will be shifted out.
 */
        SPI0CFG = 0x07;

/* how I set up the SPI0CN (bits set)
 * bit 7 -> 0 : SPIF (SPI interrupt flag)               ; CLEAR THIS!
 * bit 6 -> 0 : WCOL (write collision flag)     ; CLEAR THIS!
 * bit 5 -> 0 : MODF (mode fault flag)                  ; CLEAR THIS!
 * bit 4 -> 0 : RXOVRN (receive overrun flag)   ; CLEAR THIS!
 * bit 3 -> 0 : TXBSY (transmit busy flag), cleared by hardware
 * bit 2 -> 0 : SLVSEL (slave selected flag) set to 1 whenever device is enabled as a slave
 * bit 1 -> 1 : MSTEN (master enable)
 * bit 0 -> 1 : SPIEN (SPI enable mode)
 */
        SPI0CN  = 0x03;

    CS_LED = 1; // drive the CS line for the LEDs high (default mode)
    write_spi_led(0x02,0x00); // set LED brightness to minimum
    badge_led_grid_init();
#endif

        //Turn on transmitter
        CTR1=1;
        CTR0=0;

        //Blank variables
        gyrox.word = gyroy.word = gyroz.word = 0;
        accelx.word = accely.word = accelz1.word = accelz2.word = 0;
        pause=in_cycle=dirty=0;
}


void setMux(unsigned char in) {
#define MUX_LOOPS 60
        int i;

        A0 = in&0x01;
        A1 = (in&0x02)>>1;
        A2 = (in&0x04)>>2;
        for(i=0;i<MUX_LOOPS;i++);
}

void setCapMux(int in1, int in2) {
#define CAP_MUX_LOOPS 1000
        int i;

        DIR5 = in1;
        DIR6 = in2;

        for(i=0;i<CAP_MUX_LOOPS;i++);
}
```

```
void getIMUData() {
#define PAUSE_LOOPS 75
#define PAUSE2_LOOPS 500
        int i;

        EN1=1;
        SET_ADC(1,7);//Collect data

        // Collect data from X accelerometer
        setMux(5);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(accelx);

        // Collect data from Y accelerometer
        setMux(0);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(accely);

        // Collect data from Z1 accelerometer
        setMux(3);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(accelz1);

        // Collect data from Z2 accelerometer
        setMux(4);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(accelz2);

        // Collect data from X gyro
        setMux(1);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(gyrox);

        // Collect data from Y gyro
        setMux(2);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(gyroy);

        // Collect data from Z gyro
        setMux(6);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(gyroz);

        CLEAR_ADC(1,7);
        EN1=0;
}

void getFSRData() {
        int i;
        unsigned char temp1, temp2;

        EN2 = 1;
        SET_ADC(1,7)

        setMux(0);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(bend1);

        setMux(1);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(bend2);

        setMux(2);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr1);

        setMux(3);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr2);

        setMux(4);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr3);

        setMux(5);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr4);

        setMux(6);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr5);

        setMux(7);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(fsr6);

        // <DM> grab the digital value off of DIR5 and DIR6
        temp1 = DIR5 == 1;
        dir5_switch.byte[0] = 0x00;
        dir5_switch.byte[1] = (DIR5)?0x01:0x00;

        temp2 = DIR6 == 1;
        dir6_switch.byte[0] = 0x00;
        dir6_switch.byte[1] = (DIR6)?0x01:0x00;

        CLEAR_ADC(1,7);
        EN2=0;
}
```

```
void getCAPData() {  //ADDED <---------------------------
int i;

        SET_ADC(2,6); //(2,6) for LVL;  (2,4) for AMP LVL

        //Sensor 2
        setCapMux(1,0);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(cap2);

        //Sensor 3
        setCapMux(0,0);
        ADBUSY=1;
        while(ADBUSY); // Wait for operation to finish
        for(i=0;i<PAUSE_LOOPS;i++);
        STORE_ADC(cap3);
        CLEAR_ADC(2,6);
}


void trans3Nibbles(unsigned char high, unsigned char low)
{
        // DC Balance each six bit block
        unsigned char in,out;

        in = low&0x3F; //first six bits
        out = lookup[in];
        SEND(out);
        in = (low>>6 | high<<2)&0x3F;
        out = lookup[in];
        SEND(out);
}

void transmitIMUData(void)
{
        //Send accel data
        trans3Nibbles(accelx.byte[0],accelx.byte[1]);
        trans3Nibbles(accely.byte[0],accely.byte[1]);
        trans3Nibbles(accelz1.byte[0],accelz1.byte[1]);

        //Send gyro data
        trans3Nibbles(gyrox.byte[0],gyrox.byte[1]);
        trans3Nibbles(gyroy.byte[0],gyroy.byte[1]);
        trans3Nibbles(gyroz.byte[0],gyroz.byte[1]);
}

void transmitFSRData() {
        //Send analog data
        trans3Nibbles(fsr1.byte[0],fsr1.byte[1]);
        trans3Nibbles(fsr2.byte[0],fsr2.byte[1]);
        trans3Nibbles(fsr3.byte[0],fsr3.byte[1]);
        trans3Nibbles(fsr4.byte[0],fsr4.byte[1]);
        trans3Nibbles(fsr5.byte[0],fsr5.byte[1]);
        trans3Nibbles(fsr6.byte[0],fsr6.byte[1]);
        trans3Nibbles(bend1.byte[0],bend1.byte[1]);
        trans3Nibbles(bend2.byte[0],bend2.byte[1]);

        // transmit the state of the switch
        trans3Nibbles(dir5_switch.byte[0],dir5_switch.byte[1]);
        trans3Nibbles(dir6_switch.byte[0],dir6_switch.byte[1]);
}

void delay_iters(unsigned int num_iter)
{
        while (num_iter-- > 0);
}

void mega_delay(unsigned int num_delays)
{
        int jj;
        for (jj=0; jj<num_delays; jj++) {
                delay_iters(0xFFFF);
        }
}

void LED_swirl(int pause)
{
        disc_led_grid_draw_screen(0xFF,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0x00, 0xFF, 0x00, 0x00, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0x00, 0x00, 0xFF, 0x00, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0x00);
                delay_iters(pause);
                badge_led_grid_clear();
        disc_led_grid_draw_screen(0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF);
                delay_iters(pause);
                badge_led_grid_clear();
}

//---------------------------------------------------------------------------
// MAIN Routine
//---------------------------------------------------------------------------
void main(void)
{
    unsigned char i,j, temp, temp2, temp3;
    unsigned char pages[8] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
        unsigned char leds_are_on = 0x00;

        init(); // initialize everything
        transmit_data = 0x01; // default is to transmit data..

#ifdef DO_LEDS
    badge_led_grid_intensity(0x80);
#endif

        while(1) // spin forever
        {

#ifdef DO_DATACOLLECT

                START_TIMER(2); // So that it doesn't interrupt the first cycle

                // Get data
                getIMUData();
                getFSRData();
```

```
#ifdef DO_BIDIRECTIONAL
// this code puts the micro into a call/response mode

                SET_REC;

                // the following loop waits for 0x67, 0x6D to be received from the
                // basestation/computer, requesting data
                if (RI)
                        RI = 0;
                        temp = temp2;
                        temp2 = temp3;
                        temp3 = SBUF;
                        if (temp == 0x67 && temp2 == 0x6D && temp3 == 0x00) {

                }
#endif

                // SET_REC;
                // SCON = UART8BIT | RECEIVEON;
        if (transmit_data) {
                TI=1;
                SET_TRANS;

                SEND(0xFF); // 255
                SEND(0x00); // 0
                SEND(0xCC); // 204
                SEND(0x33); // 51
                SEND(0x66); // 102
                SEND(0x99); // 153
                SEND(0x62); // 98
                SEND(0x6d); // 109

                transmitIMUData();
                transmitFSRData();

                SEND(0x33); // 51
                SEND(0x66); // 102
                SEND(0x33); // 51
                SEND(0x66); // 102
                SEND(0x33); // 51
                SEND(0x66); // 102
                SEND(0x33); // 51
                SEND(0x66); // 102
                SEND(0x33); // 33
                SEND(0x66); // 102

                // send a line-break
                SEND('\r');
                SEND('\n');
        }

#endif

#ifdef DO_LEDS

        // visual feedback related to trigger button
if (fsr5.word < curr_pressure_threshold) {
        if (!leds_are_on) {
                all_leds_on();
                leds_are_on = 0x01;
                curr_pressure_threshold = HIGH_PRESSURE_THRESHOLD;
        }
} else {
        if (leds_are_on) {
                all_leds_off();
                leds_are_on = 0x00;
                curr_pressure_threshold = LOW_PRESSURE_THRESHOLD;
        }
}

#endif

 } // main while loop
} // main function


//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// UART0_ISR
//-----------------------------------------------------------------------------
void UART0_ISR (void) interrupt 4
{
        // we don't care about transmit-related interrupts
        if (!RI) return;

        transmit_data = SBUF;
        RI = 0; // clear the received byte flag
}
```

```
/* thesis.h
 * --------
 * Author:   David Merrill
 * Modified: February 2004
 */

#ifndef __THESIS_H__
#define __THESIS_H__

sbit CS_OB              = P3^3; // CS line for the output board microcontroller (DIR7)
sbit CS_LED             = P3^5; // CS line for the LEDs (DIR8)

void delay_iters(unsigned int num_iter);
void mega_delay(unsigned int num_delays);


#endif
```

```
/* 206.h
 * -----
 * Author: Ari Benbasat and Stacy Morris
 * Stab at a collection of useful macros for the
 *                                    ADuC812 microcontroller.
 */

/* Include SFR and sbit definitions */
#include <c8051f200.h>
#include 206SFR.h

// Structures and unions
typedef union {
        unsigned int  word;
        unsigned char byte[2];
} lu;

//ADC Stuff
#define SET_ADC(port,pin)       AMX0SL |= (port<<3)|pin;P##port##MODE &=~(1<<pin);
#define STORE_ADC(un)           un.byte[1]=ADC0L;un.byte[0]=ADC0H;
#define CLEAR_ADC(port,pin) AMX0SL =0X20;P##port##MODE |=(1<<pin);

// RS232
/* Blocking send. Check to make sure that the previous send isn't still
   going, Reset TI bit, send. */
#define SEND(x) while(!TI);TI=0;SBUF=x;
/* Non-blocking send. Check if previous send is done, if so, send */
//#define PUT(x)  if(TI) {TI=0;SBUF=x;}
#define PUT(x)


// Basic Timer Stuff
#define START_TIMER(num) TR##num=1;
#define STOP_TIMER(num)         TR##num=0;
#define SET_TIMER(num,high,low) TL##num=low;TH##num=high;
#define STORE_TIMER(num,un) un.byte[0]=TH##num;un.byte[1]=TL##num;
#define SET_TIMER2_RELOAD(high,low) RCAP2H=high;RCAP2L=low;
#define RELOAD_TIMER2() TH2=RCAP2H;TL2=RCAP2L;

// More timer stuff
// This is not really the place for the PUTs, but there is no other way.
#define TIMER_RISE_TO_FALL(num,bitname,time) \
                timeout=time; \
                STOP_TIMER(num); \
                SET_TIMER(num,0x00,0x00); \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}      \
                if(timeout!=0) {timeout=time;} \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==1){break;}}      \
                START_TIMER(num);    \
                if(timeout!=0) {timeout=time;} \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}      \
                STOP_TIMER(num); \
                if(timeout==0) {SET_TIMER(num,0x00,0x00);}
#define TIMER_RISE_TO_FALL_NOTIMEOUT(num,bitname,time) \
                STOP_TIMER(num); \
                SET_TIMER(num,0x00,0x00); \
                PUT(0x55); \
    while(bitname==1);      \
                PUT(0x55); \
    while(bitname==0);      \
                START_TIMER(num);    \
                PUT(0x55); \
    while(bitname==1);      \
                STOP_TIMER(num);
#define TIMER_RISE_TO_RISE(num,bitname,time) \
                timeout=time; \
                STOP_TIMER(num); \
                SET_TIMER(num,0x00,0x00); \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}      \
                if(timeout!=0) {timeout=time;} \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==1){break;}}      \
                START_TIMER(num);    \
                if(timeout!=0) {timeout=time;} \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}      \
                if(timeout!=0) {timeout=time;} \
                PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==1){break;}}      \
                STOP_TIMER(num); \
                if(timeout==0) {SET_TIMER(num,0x00,0x00);}
#define TIMER_RISE_TO_RISE_NOTIMEOUT(num,bitname,time) \
          STOP_TIMER(num); \
                SET_TIMER(num,0x00,0x00); \
                PUT(0x55); \
    while(bitname==1);      \
                PUT(0x55); \
    while(bitname==0);      \
                START_TIMER(num);    \
                PUT(0x55); \
    while(bitname==1);      \
                PUT(0x55);    \
                while(bitname==0);    \
                STOP_TIMER(num);
```

Figure C-3: 206.h embedded C code

```
/* output_leds.h
 * ------------
 * Author:   David Merrill
 * Modified: February 2004
 * Borrows from code written by Mat Laibowitz
 *
 */

#ifndef OUTPUT_BOARD_H
#define OUTPUT_BOARD_H

#define   BLINK_FAST    1
#define   BLINK_SLOW    0

void all_leds_on();
void all_leds_off();

void led_on(unsigned char which_led, unsigned char *pages, unsigned char send_command);
void led_off(unsigned char which_led, unsigned char *pages, unsigned char send_command);
void badge_led_grid_init (void);
void badge_led_grid_intensity (unsigned char intensity);
void badge_led_grid_enable (unsigned char en);
void badge_led_grid_clear (void);
void disc_led_grid_draw_screen (unsigned char row1, unsigned char row2, unsigned char row3,
                                unsigned char row4, unsigned char row5, unsigned char row6,
                                unsigned char row7, unsigned char row8);
void disc_led_grid_draw_screen_2 (unsigned char row1, unsigned char row2, unsigned char row3,
                                  unsigned char row4, unsigned char row5, unsigned char row6,
                                  unsigned char row7, unsigned char row8);
void badge_led_grid_blink (unsigned char on_off, unsigned char speed);
void disc_led_grid_draw_screen (unsigned char row1, unsigned char row2, unsigned char row3,
                                unsigned char row4, unsigned char row5, unsigned char row6,
                                unsigned char row7, unsigned char row8);
void badge_led_grid_draw_row (unsigned char row_num, unsigned int row_data);
void badge_led_grid_draw_row_2 (unsigned char row_num, unsigned int row_data);
void write_spi_led (unsigned char byte1, unsigned char byte2);
void sendbyte (unsigned char value);
unsigned char rol(unsigned char in,unsigned int count);
unsigned char ror(unsigned char in,unsigned int count);
#endif
```

Figure C-4: output_leds.h embedded C code

```
/* output_leds.c
 * ------------
 * Author:   David Merrill
 * Modified: February 2004
 * Borrows from code written by Mat Laibowitz
 *
 */

#include <c8051F200.h>
#include output_leds.h
#include thesis.h

unsigned char rol(unsigned char in,unsigned int count);
unsigned char ror(unsigned char in,unsigned int count);
void write_spi_led(unsigned char byte1, unsigned char byte2);
unsigned char reverse_byte(unsigned char in);

// Sets up the configuration registers in the LED driver chip
void badge_led_grid_init (void)
{
  write_spi_led(0x01,0x00); // disable decode (so that it doesn't do the digit behavior)
  write_spi_led(0x02,0x00); // set intensity to lowest
  write_spi_led(0x03,0x07); // set scan limit to 7, so that digits 0-7 are displayed
  write_spi_led(0x04,0x01); // configuration register : set to normal operation mode
  badge_led_grid_clear();   // clear the grid
}

void badge_led_grid_intensity (unsigned char intensity)
{
  write_spi_led(0x02,intensity);
}

void badge_led_grid_enable (unsigned char en)
{
  write_spi_led(0x04,en);
}

void badge_led_grid_clear (void)
{
  write_spi_led(0x04,0x21);
}

void badge_led_grid_blink (unsigned char on_off, unsigned char speed)
{
  if (on_off == 0) {
    write_spi_led(0x04,0x01);
  } else {
    if (speed == BLINK_FAST) {
      write_spi_led(0x04,0x0D);
    } else {
      write_spi_led(0x04,0x09);
    }
  }
}

void all_leds_on() {
        disc_led_grid_draw_screen(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF);
}

void all_leds_off() {
        disc_led_grid_draw_screen(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00);
}

void disc_led_grid_draw_screen (unsigned char row1, unsigned char row2,
                                unsigned char row3, unsigned char row4,
                                unsigned char row5, unsigned char row6,
                                unsigned char row7, unsigned char row8)
{
  unsigned char temp=0;

  // first need to re-arrange row2 and row5
  //row2 = reverse_byte(row2);
  //row5 = reverse_byte(row5);

  // now, write it out to the SPI
  write_spi_led(0x60,row1);
  write_spi_led(0x61,row2);
  write_spi_led(0x62,row3);
  write_spi_led(0x63,row4);
  write_spi_led(0x64,row5);
  write_spi_led(0x65,row6);
  write_spi_led(0x66,row7);
  write_spi_led(0x67,row8);
}

void disc_led_grid_draw_screen_2 (unsigned char row1, unsigned char row2,
                                  unsigned char row3, unsigned char row4,
                                  unsigned char row5, unsigned char row6,
                                  unsigned char row7, unsigned char row8)
{
  unsigned char temp=0;

  write_spi_led(0x40,row1);
  write_spi_led(0x41,row2);
  write_spi_led(0x42,row3);
  write_spi_led(0x43,row4);
  write_spi_led(0x44,row5);
  write_spi_led(0x45,row6);
  write_spi_led(0x46,row7);
  write_spi_led(0x47,row8);
}

// reverse the bit order in a byte
unsigned char reverse_byte(unsigned char in)
{
        int i;
        unsigned char temp=0;

        for (i=0; i<8; i++) {
                temp |= ((0x01 << i) & in) << 7-i;
        }
        return(temp);
}

// rotate a byte in place
unsigned char rol(unsigned char in,unsigned int count)
{
  return (in << count | in >> (8-count));
}

unsigned char ror(unsigned char in,unsigned int count)
{
  return (in >> count | in << (8-count));
}

void get_mask_and_idx(unsigned char which_led, unsigned char *mask_byte,
                                                unsigned char *page_idx) {
        unsigned char which_bit, OR_mask;

        // figure out which page it's in
        *page_idx = which_led/8;
```

```
        // figure out which bit it is
        which_bit = which_led%8;

        // generate the mask to OR with that byte to turn on the
        // appropriate bit
                if (*page_idx == 1 || *page_idx == 4) {
                        OR_mask = 0X80;
                        OR_mask = rol(OR_mask,which_bit);
                } else {
                        OR_mask = 0X40;
                        OR_mask = ror(OR_mask,which_bit);
                }

                *mask_byte = OR_mask;
}

void led_on(unsigned char which_led, unsigned char *pages,
                        unsigned char send_command) {
        unsigned char which_page, OR_mask;

        get_mask_and_idx(which_led, &OR_mask, &which_page);
        pages[which_page] |= OR_mask;
        if (send_command) disc_led_grid_draw_screen(pages[0],pages[1],
                pages[2],pages[3],pages[4],pages[5],pages[6],pages[7]);
}

void led_off(unsigned char which_led, unsigned char *pages,
                         unsigned char send_command) {
        unsigned char which_page, which_bit, OR_mask;

        get_mask_and_idx(which_led, &OR_mask, &which_page);
        pages[which_page] &= ~OR_mask;
        if (send_command) disc_led_grid_draw_screen(pages[0],pages[1],
                pages[2],pages[3],pages[4],pages[5],pages[6],pages[7]);
}

void write_spi_led (unsigned char byte1, unsigned char byte2)
{
        CS_LED = 0;         // assert CS

        while(TXBSY);       // wait for transmit buffer to be free
        SPIOCN = 0X03;      // clear the warning flags
        SPIODAT = byte1;    // write byte to output buffer

        while(TXBSY);       // wait for transmit buffer to be free
        SPIOCN = 0X03;      // clear the interrupt flag
        SPIODAT = byte2;    // write byte to output buffer

    while(TXBSY);           // wait for transmit buffer to be free
        CS_LED = 1;         // dis-assert CS
}
```

Figure C-5: output_leds.c embedded C code

Appendix D

Java code summary

This appendix contains names and descriptions of all Java classes (other than the classes in the Java Runtime Environment) written for, and used in this project.

class AMCGUIStateSaver

Description: Keeps an ArrayList of VarianceReport objects, and an ArrayList of MappingData objects. This class is for saving state to disk, and is instantiated at serialization-time.

interface ButtonResponder

Description: This interface defines the behavior that a class must implement in order to respond to button presses of the toggle button and the trigger button on the device.

class ButtonWatcher

Description: Registered as a StackPacketListener, this class pulls out the digital and trigger button state from each StackPacket that comes in, and messages a ButtonResponder object with relevant changes in state. This is where the real "dispatch" of button commands happens.

class ChartPanelHandler

Description: Registered as a StackPacketListener, this class broadcasts incoming stack packets to all ChartPanel objects that it has references to. This is where the data dispatch to the StripChart objects in the GUI takes place.

class CommonStackDataFilter

Description: A stack data filter object that handles the "common case", this class is instantiated with an array of "flags". These flags are the bytes that are expected (in the order given) to proceed each valid data packet from the stack in the serial data stream. For instance, in the current implementation, the hex bytes 0x62,0x6D proceed each batch of sensor data. This class assembles a complete StackDataPacket object from the serial data stream, and passes the completed packet to all registered listeners.

interface DataHandler

Description: This interface defines the behavior that any class handling incoming serial data from a stack should implement. It consists of a single function, newData.

class DataLogger

Description: Handles the writing of log data to a log file. Timestamps each log entry to millisecond resolution.

Figure D-1: Main program Java package: edu.mit.media.amc

class DataLoggerManager

Description: A wrapper around the DataLogger class, the DataLoggerManager handles the initialization of the DataLogger, and provides a function to generate a unique directory name - which is useful when running a user study so that each user's data can be kept in a separate directory.

class DataStreamCharacterizer

Description: Watches an incoming data stream, and can be queried for information about the data, such as instantaneous windowed variance, max variance, max/min values, etc.

class DTW

Description: Pure Java implementation of the Dynamic Time-Warping algorithm. Contributors to this code (originally written in C) include Malcom Slaney, Tony Robinson, and Yuri Ivanov.

class DTWPureJavaStackGestureRecognizer

Description: Implements the StackGestureRecognizer interface to manage the state associated with gesture training. Holds the reference to the data for all of the trained gestures, and can be messaged to start/stop data collection for training or classification.

class DTWStackGestureRecognizer

Description: NOT USED IN FINAL VERSION. Concrete implementation of the StackGestureRecognizer interface, that uses Dynamic Time-Warping. Passes all data into the C .dll for storage, and makes DTW calls to the C .dll.

class FlexibleMappingManager

Description: Manages the creation and clearing of input-DOF to effect parameters. Also provides functions that can enable/disable all effects.

abstract class FlexibleMappingObject

Description: A FlexibleMappingObject keeps track of all of the data relevant to a sensor-to-effect parameter mapping. Designed with OSC communication in mind rather than MIDI, this class keeps track of the mapping's symbol, polarity (which end of the input's range is mapped to which end of the output's range), and the sensor's VarianceReport (this includes information about the max/min of the sensor's range, and is used to scale incoming values to a 0-1 range).

class FlexibleMIDIMappingObject

Description: Subclass of FlexibleMappingObject that is MIDI-specific.

class FlexibleOSCMappingObject

Description: Subclass of FlexibleMappingObject that is OSC-specific.

class GestureDataManager

Description: Keeps an array of gesture models and examples, and implements the state machine that defines how new sensor data should be handled (i.e. should it be put into a gesture-class example?, stored in the current SingleGestureData object for subsequent classification?, etc..)

class GestureModelsInfo

Description: Keeps track of how many gesture models each class has, and keeps an array of SingleGestureInfo objects. This is a state-saver class that is created at serialization-time.

interface Looper

Description: Interface which defines the messages that any Looper for this device will have to implement. Functionality is based on having a toggle button on the device that indicate "up", "down" and "in".

class LooperStack

Description: Stack-based sound-looping manager that implements the Looper interface. Manages the current set of sounds that are being looped, and allows a user to loop a sound, un-loop a sound, and cancel all looping.

class MappingData

Description: Holds the symbol and polarity of a single mapping. This is a state-saver class that is created at serialization-time.

class MappingLearner

Description: Watches the variance on all of the sensors while the OutputControlMessageSweeper (OCMS) is sweeping a parameter sinusoidally. Registered as a listener for "extrema" messages from the OCMS, at each extrema it notes the (a) sensor exhibiting the maximal variance, and (b) which end of its dynamic range that sensor is closest to. If the MappingLearner sees a consistent pattern of excitation (i.e.

same sensor, same polarity) over a given number of consecutive extrema, a mapping is created via the FlexibleMappingManager object.

interface MaxMinListener

Description: Interface that defines the behavior that a class must implement in order to be a listener registered to receive notification when a running OutputControlMessageSweeper object has reached an extrema. Defines a single function, maxOrMinReached.

class MidiDeviceManager

Description: Wrapper around the Java MIDI functionality which simplifies the querying aspects of the MIDI subsystem, and the sending of MIDI messages.

class MIDIOutputControlMessage

Description: Subclassed from OutputControlMessage, MIDIOutputControlMessage knows how to send a MIDI control message

class NonLinearNumberMapper

Description: Built to allow nonlinear mappings from input DOF to output effect modulation, this class did not end up being used.

interface NumberMapper

Description: Interface that defines the behavior that a class must implement in order to be considered a NumberMapper. The purpose of a NumberMapper is to map an input number onto an output number. This interface defines a single function, mapNumber, and allows any arbitrary mapping of input sensor value to output effect control value.

class OSCOutputControlMessage

Description: Subclassed from OutpuControlMessage, OSCOutputControlMessage knows how to send an OSC control message

abstract class OutputControlMessage

Description: Abstract class which contains the data for a single control message. Holds a data value and a String symbol.

class OutputControlMessageRouter

Description: OutputControlMessageRouter runs in its own thread and loops endlessly, dequeueing OutputControlMessages in a synchronized way from a queue, and

transmitting them to the outside world. This is the bottleneck for all outgoing control messages to a synthesizer, and for that reason it is also the class that does much of the logging for experimental purposes.

class OutputControlMessageSweeper

Description: Creates a sinusoidally oscillating parameter trajectory to be used in affordance mapping. Sends an optional start/stop signal to trigger a particular sound to play for the duration of the parameter sweep.

class Queue

Description: A Queue implementation that uses the java Vector class. This is based on some code that I found online.

interface SensorDataView

Description: Interface that defines the behavior that a class must implement in order to be considered a SensorDataView. Defines two versions of the newData function by which the SensorDataView receives a single new datum for storage and/or display.

class SimpleMidiMapper

Description: Permits the simple assignment of indices (zero-indexed integers) to 3-byte MIDI messages, and the playing of the messages.

class SingleGestureData

Description: Keeps track of the samples for a single gesture. Contains methods for reading a gesture data file from disk, and writing a single gesture data file out to disk.

class SingleGestureInfo

Description: This is a state-saving class that is instantiated at serialization-time to keep track of the data associated with a single gesture class. Tracks the presence of a class model, the length of the model, the number of gesture examples in that class, the lengths of the examples, the filename of the model (if present), and the filenames of each of the gesture examples.

class SingleGestureModelData

Description: Stores data for a single gesture class. Keeps a model gesture, and an array of examples.

class SmartButtonGroup

Description: This class extends javax.swing.ButtonGroup, adding the capability for cycling through a list of buttons modulo the number of buttons (i.e. when you try to go "down" past the bottom of the list to select the next button, you end up at the top again). Provides facilities for selecting next and previous buttons from a list. This class is used in the GUI for the list of possible sounds.

class SoundTriggerManager

Description: Manages the starting and stopping of sample playback.

abstract class StackDataFilter

Description: Abstract class meant to provide some common functionality for specific StackDataFilter subclasses. A StackDataFilter is responsible for parsing the stream of data coming from a Stack into complete data packets, then passing each complete packet to each of the registered listeners.

class StackDataPacket

Description: Class for storing a packet of data from the Stack.

class StackDataPacketInfo

Description: Immutable class for storing configuration info about stack data packets. Is meant to be used in a static way, this class defines defaults specific to my particular stack and device configuration.

class StackDeviceHandler

Description: Manages the serial connection to the stack, including opening and closing of the serial channel, and registering/unregistering listeners for data coming from the stack.

interface StackGestureRecognizer

Description: This interface defines behavior that any class that manages data-receiving and gesture-recognition activities will have to implement.

interface StackPacketListener

Description: This interface defines the behavior that any class wishing to be a listener for Stack packets will have to implement. It consists of a single function, newStackPacket.

class StackPacketParser

Description: Separates the data from a single packet, feeding the individual data to the appropriate views. Views register as listeners for a particular index of incoming stack packets. For instance, StackPacketParser is used to distribute sensor data to the StripCharts, since they each only show data from a single sensor.

interface SerialClient (written by DJM)

Description: This interface defines the behavior that a SerialClient must implement. Defines the portClosed method, as well as two versions of handleNewSerialData.

class SerialConnection (written SUN, modified by DJM)

A class that handles the details of a serial connection. Originally distributed with the communications API from Sun Microsystems, this file was modified to enqueue new serial data into a queue in a synchronized code block rather than the original behavior in which it would write the data into a StringBuffer.

class SerialConnectionException (written by SUN)

Distributed with the communications API from Sun Microsystems, this exception can be thrown in the course of serial communication.

class SerialGrabber (written by DJM)

Description: SerialGrabber runs in its own thread, and its purpose is to grab new bytes of serial data from the queue in the SerialConnection object. This queue is accessed by both SerialGrabber and SerialConnection in a synchronized manner using wait/notify, so SerialGrabber is careful to avoid any time-consuming routines within the synchronized codeblock. All that is done in the synchronized codeblock is dequeueing of the queue. Then, once outside the synchronized block, the data is passed off to any registered listeners.

class SerialParameters (written by SUN)

Distributed with the communications API from Sun Microsystems, SerialParameters holds all relevant state pertaining to a serial connection.

class serialUtilities (written by DJM)

Description: This class provides a wrapper for the CommPortIdentifier.getPortIdentifiers() method which returns an enumeration of the available ports on a system.

Figure D-2: Serial Java package: edu.mit.media.amc.serial

class ChartPanel (written by drew/ckline/ayb)

Implements a panel full of stripcharts, each of which can display data from a single sensor.
author: Andy Wilson (drew@media.mit.edu)
modified: Chris Kline (ckline@media) to application-independent
modified: Ari Benbasat (ayb@media) to add to new innards package
modified: David Merrill (dmerrill@media) such that addDataSource returns the StripChart object that has been instantiated

interface DataSource (written by drew/ckline/ayb)

Interface that defines the behavior that a class must implement in order to be used as a DataSource : specifies the method getValue.

interface EnhancedDataHandler (written by DJM)

Interface that defines the behavior that a class must implement in order to be used as a EnhancedDataHandler : specifies the method newVarianceReport, which installs a VarianceReport into the DataHandler.

interface EnhancedDataSource (written by DJM)

Description: Builds on the DataSource idea from the innards.util.widgets package, adding specifications that relate to tracking a running variance, and enabling/disabling the processing of the DataSource in order to save on CPU cycles.

class GraphPanel (written by drew/ckline/ayb)

Implements the drawing routines for plotting a single sensor data trace in time, like a software oscilloscope. StripChart is based on this class.
author: Andy Wilson (drew@media.mit.edu)
extended and augmented by: Chris Kline (ckline@media.mit.edu)
converted (as if it was hard) to innards package by AYB
modified to be enable-able/disable-able by DJM

class StripChart (written by drew/ckline/ayb)

Keeps track of the most recent N samples captured from a particular sensor, and uses the functionality defined in GraphPanel to plot the data.

Figure D-3: Sensor data display Java package: innards.util.widgets

class StripChartAutoAxes (NOT USING THIS)

A strip chart with asymmetrically smoothed axes (e.g they always get big enough but slowly shrink back to be within a certain range of the average of the data).

class VarianceReport (written by DJM)

Description: Data-keeper class for information about the variance on a particular sensor. Can be loaded with information about max/min/avg variance seen on the sensor over a window of time, as well as max/min instantaneous sample values over the window. Used by FlexibleMappingObject's to scale incoming data to lie in the [0-1] range, and by MappingLearner to determine which sensor experienced the most variance since the last poll.

class JDirectoryChooser (written by David Ekholm)

A fix to JFileChooser that correctly allows only directories to be selected.
Does the trick by accepting "." as selection of the currently displayed directory. David Ekholm authored this class, and it is available online as part of his JAlbum project. See http://jalbum.net/ for details.

Figure D-4: DirectoryChooser Java package: se.datadosen

Appendix E

Experimental materials

Subject Instructions

1) Watch the two instructional videos to get an understanding of how to find sounds, train gestures, and make effect associations.

2) Train and test some gestures. (in the ControlPanel tab, with the GESTURE TRAINING button depressed for training, and the GESTURE TESTING button depressed for testing). Try looping and un-looping a sound. Click the **SAVE** button.

3) Make some input-to-effect associations (you can also test your associations with the "TEST" button underneath each large button). Click the **SAVE** button.

4) Click back to the ControlPanel and click the PLAY button. Spend as much time as you like playing with the system (triggering sounds, looping them, applying effects, un-looping sounds, etc..). The experimenter will stop you when it is time to move on to the next part of the experiment, or if you feel that you are finished, please alert the experimenter.

Figure E-1: Instruction page given to subjects in the Training condition

Subject Instructions

1) Watch the two instructional videos to get an understanding of how the system works.

2) Trigger each of the sounds at least once, and try the looping feature.

3) When you have one or more of the sounds looping, try out each of the effects.

4) Spend as much time as you like playing with the system (triggering sounds, looping them, applying effects, un-looping sounds, etc..). The experimenter will stop you when it is time to move on to the next part of the experiment, or if you feel that you are finished, please alert the experimenter.

Figure E-2: Instruction page given to subjects in the Presets condition

This video will show you how to train the system to recognize your own gestures to trigger sounds, and your own manipulations to modify the triggered sounds.

To begin, click the "GESTURE TRAINING" button in the "ControlPanel tab. This will put you into gesture training mode.

Now, you can use the thumb toggle button to cycle up or down through the collection of sounds.

When you find a sound that you like, you can train a gesture. This will associate your gesture with the sound, like a "gestural bookmark" so that when you are playing the instrument, you can trigger the sound with the gesture.

When executing a gesture, you must squeeze and hold the trigger button through the entire gesture. Then, let go once you're done.

The lights around the circle will show you when the device is listening to your gesture.

The device can sense movement and rotation in any direction, so your gesture should be some combination of movement and rotation.

Remember. Squeeze and hold the trigger button through your entire gesture. Then let go.

After training a gesture, you can click on the "GESTURE TESTING" button to try it out. Triggering a gesture works exactly like training the gesture. Squeeze the trigger button, do the gesture, then let go of the trigger button. You can go back and forth between training and testing as many times as you like.

When you've trained as many gestures as you want, click on the save button to save your gestures.

Now, click on the button labeled "tweak mapping", and click on the tab labeled "TestPanel"

Here you can associate the squeezing and twisting inputs of the device with sound effects. Each orange button is a different sound effect.

The way it works is this : When you click on the orange button, the system starts sweeping that effect from its minimum to its maximum, and watching for activity on the inputs.

Figure E-3: "Gesture and Manipulation Training" user study script (Training)

This video will demonstrate the force and twist-sensitive inputs that are part of the gestural music controller.

This button can sense the continuous pressure you apply to it.

This handle can sense the pressure that you squeeze it with.

The device can sense how it is being twisted.

These buttons can sense the continuous pressure you apply to them.

These inputs will be associated with effect parameters, and they allow you to shape the sounds that you trigger with the device.

Figure E-4: "Affordance Training" user study script (Training)

This movie will show you which gestures you can use to trigger sounds with the system.

It's important to remember that when executing a gesture, you must squeeze and hold the trigger button through the entire gesture. Then, you can let go when you are done.

Here is the first gesture

Here is the second gesture

Here is the third gesture

Push the toggle button up to put the most recent sound you triggered into looping mode. You can cancel the current looping behavior by pressing down on the same button. Press the button directly in to cancel all looping.

Figure E-5: "Preset Gestures" user study script (Presets)

This video will show you how to apply effects to the sounds that you trigger.

Twisting the device like this applies a sweeping filter to the sound.

Squeezing on this button applies a tremolo effect to the sound.

Squeezing on the handle like this applies a pitch shift to the sound.

Figure E-6: "Preset Mappings" user study script (Presets)

Pre-Interaction Questions

Answer these questions first, before starting any other part of the experiment.

* 1. What is your experiment ID code? (the experimenter should give you this) []

* 2. What is your age? []

* 3. What is your gender?

* 4. How experienced are you at playing any musical instrument (other than singing)? As you answer this question, consider of the instrument, if any, that you are most familiar with.

(1) not experienced at all (2) (3) (4) (5) (6) (7) very experienced

5. What instrument (if any) do you have the most experience with? []

* 6. About how much time per week (on average) do you spend playing a musical instrument?

0 min 30 min 1 hr 2 hrs 5 hrs 10 hrs 20+ hrs

* 7. How much experience do you have performing any musical instrument (other than singing)? As you answer this question, consider the instrument, if any, that you are most familiar with.

(1) not experienced at all (2) (3) (4) (5) (6) (7) very experienced

8. What instrument (if any) have you performed with most?

* 9. About how much time per week (on average) do you spend performing with a musical instrument?

0 min 30 min 1 hr 2 hrs 5 hrs 10 hrs 20+ hrs

* 10. How would you rate your general level of musicianship (including singing)?

(1) novice (2) (3) (4) (5) (6) (7) expert

* 11. How much do you enjoy listening to music?

(1) not at all (2) (3) (4) (5) (6) (7) very much

* 12. About how much time per week (on average) do you spend listening to music?

0 min 30 min 1 hr 2 hrs 5 hrs 10 hrs 20+ hrs

* 13. How much do you enjoy playing music?

(1) not at all (2) (3) (4) (5) (6) (7) very much

* 14. Have you ever conducted an orchestra or led a band? How experienced are you with this task, or a similar one?

(1) not at all experienced (2) (3) (4) (5) (6) (7) very experienced

---

Figure E-7: Pre-interaction survey, filled out by all participants

Post-Part-A Questions

These questions are to be answered after you have completed part A of the study.

* 15. How enjoyable was your experience using this system?

(1) not enjoyable at all (2) (3) (4) (5) (6) (7) extremely enjoyable

* 16. How engaging was your experience using this system?

(1) not engaging at all (2) (3) (4) (5) (6) (7) extremely engaging

* 17. How expressive did you feel that you could be in using this system?

(1) not very expressive (2) (3) (4) (5) (6) (7) extremely expressive

* 18. How easy was it for you to learn to trigger the sounds that you wanted?

(1) not easy at all (2) (3) (4) (5) (6) (7) extremely easy

* 19. How easy was it for you to actually trigger the sounds successfully once you had figured out the gesture to sound associations?

(1) not easy at all (2) (3) (4) (5) (6) (7) extremely easy

* 20. How easy was it for you to learn to manipulate the effects?

(1) not easy at all (2) (3) (4) (5) (6) (7) extremely easy

* 21. How easy was it for you to actually manipulate the effects once you had figured out the manipulation to effect associations?

(1) not easy at all (2) (3) (4) (5) (6) (7) extremely easy

* 22. How likely would you be to want to use this system again in a non-performance (play) context?

(1) not likely at all (2) (3) (4) (5) (6) (7) extremely likely

* 23. How likely would you be to want to perform with this system?

(1) not likely at all (2) (3) (4) (5) (6) (7) extremely likely

* 24. How much "personalization" did you feel that this system offered to you?

(1) no personalization (2) (3) (4) (5) (6) (7) very much personalization

* 25. Please rate your feelings about the level of personalization you experienced.

(1) far too little personalization (2) (3) (4) (5) (6) (7) far too much personalization

* 26. How appropriate did you feel that the connection was between the types of gestures possible with the instrument and the sounds that could be triggered?

(1) not appropriate at all (2) (3) (4) (5) (6) (7) very appropriate

* 27. How appropriate did you feel that the connection was between the types of manipulations possible with the instrument and the effects that could be controlled?

(1) not appropriate at all (2) (3) (4) (5) (6) (7) very appropriate

* 28. How well did the system recognize your gestures?

(1) not well at all (2) (3) (4) (5) (6) (7) extremely well

* 29. How novel do you feel this system is?

(1) not very novel (2) (3) (4) (5) (6) (7) extremely novel

* 30. How in control did you feel during your experience with the system?

(1) not in control at all (2) (3) (4) (5) (6) (7) very much in control

---

Figure E-8: Post-part-A/B survey, filled out by all participants

Final Questions

These questions are the last batch that we will ask you to answer.

* 47. Please rate your general level of enjoyment in playing new musical instruments (instruments that you do not have previous experience with).

    (1) I do not enjoy this at all (2) (3) (4) (5) (6) (7) I enjoy this very much

* 48. Have you ever built your own traditional musical instrument? [Y/N]

* 49. Have you ever built your own unconventional musical instrument (electronic or otherwise)? [Y/N]

* 50. How creative do you consider yourself?

    (1) not very creative (2) (3) (4) (5) (6) (7) extremely creative

* 51. How much of a risk-taker are you?

    (1) not at all (2) (3) (4) (5) (6) (7) very much

* 52. How much do you enjoy solving open-ended problems, that have no "right" answer?

    (1) do not enjoy this at all (2) (3) (4) (5) (6) (7) enjoy this very much

* 53. How proficient are you with technology?

    (1) not at all (2) (3) (4) (5) (6) (7) very much so

* 54. Which of the two setups do you think allows for more expressivity? [first/second]

* 55. Which of the two setups do you think is easier to learn to use? [first/second]

* 56. Which of the two setups do you think is more engaging? [first/second]

* 57. Which of the two setups do you think is more enjoyable to use? [first/second]

* 58. Which of the two setups do you think has a better level of personalization? [first/second]

* 59. Which of the two setups do you think is more novel? [first/second]

* 60. Which of the two setups would you be more interested in performing with? [first/second]

61. Do you have any additional comments about the gestural control system, and how it may or may not affect, improve, or detract from your sense of engagement, expressivity, ease of learning and use, feeling of personalization, interest in performance, novelty, or anything else? []

62. Are there other types of input control that you felt were missing, or that you would like to see in this instrument? []

63. Do you have any additional comments about the experience as a whole? []

Figure E-9: Final Questions survey, filled out by all participants

Appendix F

Pure-Data patches

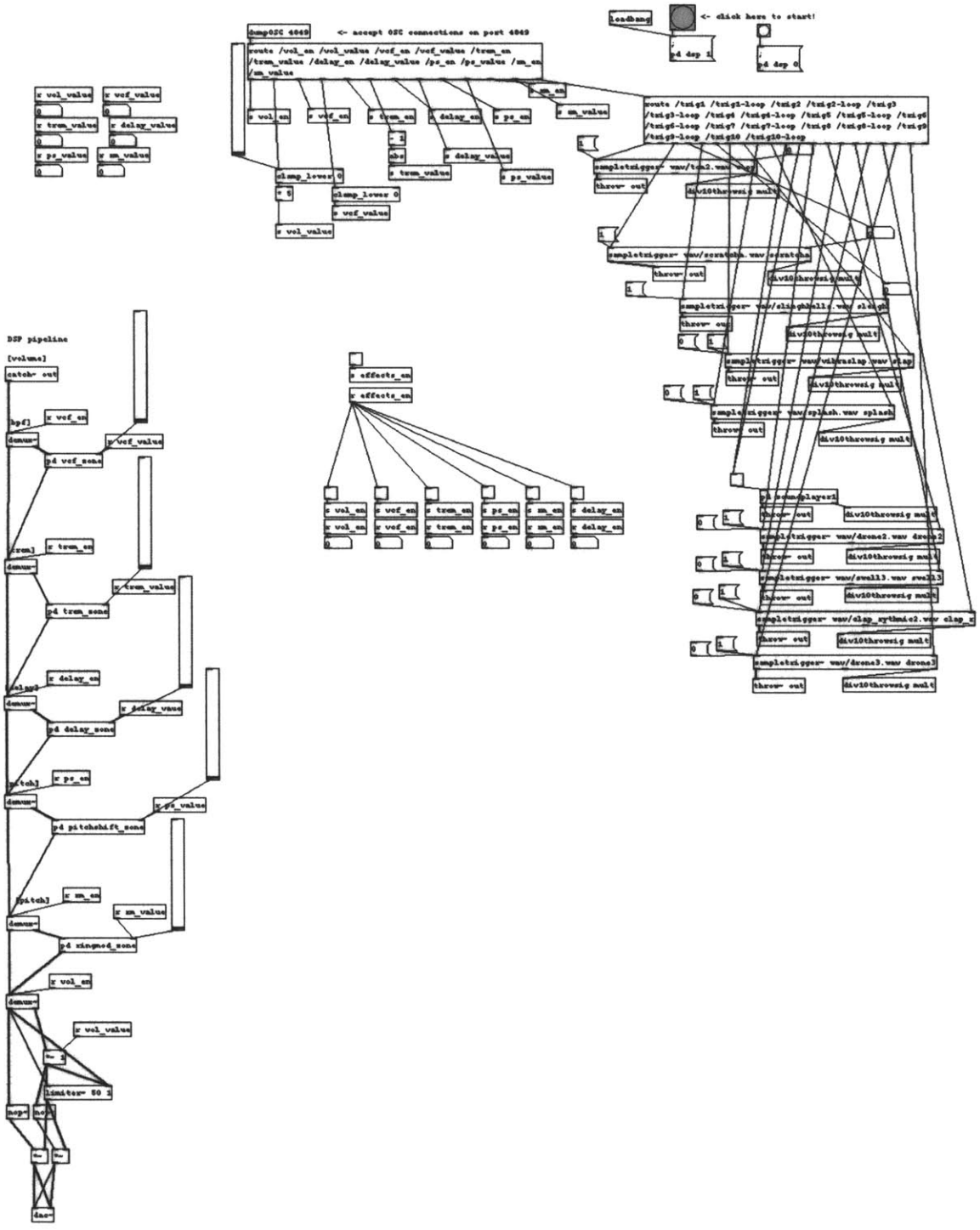

Figure F-1: The main Pure-Data patch for sound triggering and modification

Bibliography

AAAI 2000 Spring Symposium on Adaptive User Interfaces, technical report at: http://www.isle.org/~aui/papers/, Spring Symposium webpage at: http://www.aaai.org/Symposia/Spring/2000/

ActiveWindows "Microsoft SideWinder Dual-Strike" review, website at: http://activewin.com/reviews/hardware/joysticks/microsoft/dual_strike/index.shtml

Anthes, Joshua. Unique considerations for data radio UARTS. http://www.rfm.com/corp/appdata/AN43.pdf.

Bailey, Derek. website: http://www.shef.ac.uk/misc/rec/ps/efi/mbailey.html

Ballard, D. *An Introduction to Natural Computation.* MIT Press, Cambridge, MA. 1997

Bellman, R.E. *Dynamic Programming,* Princeton University Press, Princeton, New Jersey, USA, 1957.

Benbasat, Ari. "An Inertial Measurement Unit for User Interfaces," Masters of science thesis. MIT Media Lab, 2000.

Benbasat, A.Y., Morris, S.J, and Paradiso, J.A. "A Wireless Modular Sensor Architecture and its Application in On-Shoe Gait Analysis". In *Proceedings of the 2003 IEEE International Conference on Sensors*. Volume 2, pp 1086-1091. 2003

Bledsoe, W. W., and Browning, I., "Pattern Recognition and Reading by Machine", In *Proceedings of the Eastern Joint Computer Conference*. 1959.

Blumberg, Bruce et al. "Integrated Learning for Interactive Synthetic Characters". In *Proceedings of ACM SIGGRAPH.* 2002.

Boulanger, R., and Matthews, M. "The 1997 Mathews Radio-Baton & Improvisation Modes". From the *Proceedings of the 1997 International Computer Music Conference*. Thessaloniki Greece. 1997

Bowler, I., A. Purvis, P. Manning, and N. Bailey. 1990. "On Mapping N Articulation onto M Synthesiser-Control Parameters." In *Proceedings of the 1990 International Computer Music Conference*. San Francisco, International Computer Music Association, pp. 181-184.

Buchla and Associates, website at: http://www.buchla.com/

Carroll, J.M., Mack, R.L, and Kellogg, W.A. "Interface metaphors and user interface design." *Handbook of human-computer interaction*. Helander, M. (ed.) Elsevier Science Publishers B.V. (North-Holland). 67-85. 1988

Chadabe, Joel. "Devices I Have Known and Loved". In *Trends in Gestural Control of Music*. M.M. Wanderley and M. Mattier, eds. Ircam – Centre Pompidou. 2000

Clynes, Manfred. *Sentics: the Touch of the Emotions*. New York: Doubleday and Company, 1977.

Cont, A., Coduys, T., and Henry, C. "Real-time Gesture Mapping in Pd Environment using Neural Networks". In the proceedings of the *International Conference on New Interfaces for Musical Expression*. Hamamatsu 2004.

Cook, P., "Remutualizing the Instrument: Co-Design of Synthesis Algorithms and Controllers." In *Proceedings of the Stockholm Music and Acoustics Conference (SMAC)*. Stockholm, Sweden. 2003

Cope, David, "Pattern Matching as an Engine for the Computer Simulation of Musical Style". In *Proceedings of the 1990 International Computer Music Conference*. San Francisco: Computer Music Association. 1990

Dubnov, S., Assayag, G., Lartillot, O., and Bejerano, G. Using Machine-Learning Methods for Musical Style Modeling. In *IEEE Computer*, Volume 36, Issue 10. October 2003

Duda, Richard O., Hart, Peter E., Stork, David G., *Pattern Classification (2nd Edition).* Wiley-Interscience. 2000

Eslambolchilar, P., Crossan, A., and Murray-Smith, R. "Model-based Target Sonification on Mobile Devices". In the *Proceedings of the International Workshop on Interactive Sonification*. Bielefeld. 2004

Fels, S. and Hinton, G., "Glove-TalkII: An Adaptive Gesture-to-Formant Interface", In *Proceedings of Computer Human Interaction 1995* (SIGCHI95), pp. 456--463, May 7-11, Denver, CO. 1995

Frith, Fred. website: http://www.fredfrith.com/

Gadd, A., and Fels, S. MetaMuse: Metaphors for Expressive Instruments. In Proceedings of the *International Conference on New Interfaces for Musical Expression*. Dublin 2002.

Galeyev, B. "Leon Theremin---Electronic Art Pioneer," in *Soviet Faustus* (Kazan, 1995) 96 pp.

Garnett, G., and C. Goudeseune, "Performance Factors in Control of High-Dimensional Spaces". In *Proceedings of the 1999 International Computer Music Conference*. San Francisco, International Computer Music Association, pp. 268 - 271. 1999

Garton, Brad, "The Elthar Program", *Perspectives of New Music.* Vol 27. No. 1 1989.

Goudeseune, Camille. *Composing with Parameters for Synthetic Instruments.* Dissertation, University of Illinois at Urbana-Champaign. 2001.

Hunt, A., and R. Kirk. "Mapping Strategies for Musical Performance." In M. Wanderley and M. Battier, eds. *Trends in Gestural Control of Music*. Ircam - Centre Pompidou. 2000

Hunt, A., M. Wanderley, and R. Kirk. "Towards a Model for Instrumental Mapping in Expert Musical Interaction." In *Proceedings of the 2000 International Computer Music Conference. San Francisco,* International Computer Music Association, pp. 209-212. 2000

Hunt, A., Wanderly, M. and Paradis, M. "The Importance of Parameter Mapping in Electronic Instrument Design." In *New Interfaces for Musical Expression* proceedings. Dublin, 2002.

Illposed Software webpage, online at: http://www.mat.ucsb.edu/~c.ramakr/illposed/

Ishii, H., Wineski, C., Orbanes, J., Chun, B., and Paradiso, J. "PingPongPlus: Design of an Athletic-Tangible Interface for Computer-Supported Cooperative Play." In the proceedings of the *Conference on Human Factors in Computing Systems*. 1999

Ishii, Hiroshi. "The Last Farewell - Traces of Physical Presence." In *Interactions*, volume v.4. 1998

Ivanov, Y. *State Discovery for Autonomous Learning*. PhD Thesis. Massachusetts Institute of Technology. 2002

Joystick Review website:
http://www.joystickreview.com/microsoft/sidewinderfreestylepro.asp

Kaelbling, L. *Learning in Embedded Systems*. PhD Thesis, Stanford University 1990

Koza, J., Bennett, F., Andre, D., and Keane, M. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann. 1999

Kvifte, T. *Instruments and the electronic age: toward a terminology for a unified description of playing technique*. Oslo: Solum Forlag. 1989

Langley, P. (1997). "Machine learning for adaptive user interfaces". In *Proceedings of the 21st German Annual Conference on Artificial Intelligence* (pp. 53-62). Freiburg, Germany: Springer.

Lee, M., A. Freed, and D. Wessel. 1991. "Real-Time Neural Network Processing of Gestural and Acoustic Signals." In *Proceedings of the 1991 International Computer Music Conference.* San Francisco, International Computer Music Association, pp. 277-280.

Lexicon MPX110 website, online at: http://www.lexiconpro.com/mpx110/index.asp

Mahillon, V.C. 1893. *Catalogue descriptif et analytique du Musée instrumental du Conservatoire royal de musique de Bruxelles*. Brussels : Th. Lombaerts.

Mandelis, J., "Genophone: An Evolutionary Approach to Sound Synthesis and Performance," *Proceedings ALMMA 2001: Artificial Life Models for Musical Applications Workshop*, Prague, Czech Republic: Editoriale Bios, pp. 37-50, 2001.

Manor, Justin. http://acg.media.mit.edu/people/manor/fabrique.html

Marrin, T. *Toward an Understanding of Musical Gesture: Mapping Expressive Intention with the Digital Baton*. MS Thesis. MIT. 1996

Marrin, T. and Paradiso, J. "The Digital Baton: a Versatile Performance Instrument." *International Computer Music Conference Proceedings*. Thessaloniki, Greece, 1997.

Marrin, T. and Picard, R. "The Conductors Jacket: a Testbed for Research on Gestural and Affective Expression." Presented at the *XII Colloquium for Musical Informatics*, in Gorizia, Italy, September 1998.

Mitchell, T., *Machine Learning*. McGraw Hill, 1997

Modler, P., Myatt, T., and Saup, M. "An Experimental Set of Hand Gestures for Expressive Control of Musical Parameters in Realtime". *New Interfaces for Musical Expression* proceedings. Montreal, 2003.

Modler, P. 2000. "Neural Networks for Mapping Gestures to Sound Synthesis." In M. Wanderley and M. Battier, eds. *Trends in Gestural Control of Music*. Ircam - Centre Pompidou.

National Museum of Music website "Images from the Rawlins Gallery" page, at http://www.usd.edu/smm/Cellos/Amati/Amaticello.html

Needleman, S.B., and Wunsch, C.D. "A general method applicable to the search for similarities in the amino acid sequences of two proteins." *1970 Journal of Molecular Biology* 48:443-453.

Norman, D. A. *The Psychology of Everyday Things*. New York: Basic Books. 1988.

OpenSound Control Home Page, online at:
http://www.cnmat.berkeley.edu/OpenSoundControl/

Ostertag, Bob. "Human Bodies, Computer Music". In *Leonardo Music Journal*. Journal 12. 2002.

O'Modhrain, Sile. *Playing by Feel: Incorporating Haptic Feedback into Computer-Based Musical Instruments*. PhD Dissertation. Stanford University. 2000

Pachet, Francois, "The Continuator: Musical Interaction with Style", In *ICMA*, editor, Proceedings of ICMC, pages 211-218, September 2002. ICMA.

Paradiso, Joseph A., "The Brain Opera Technology: New Instruments and Gestural Sensors for Musical Interaction and Performance", *Journal of New Music Research*. Vol 28. No 2 (June) 1999.

Paradiso, J., and Gershenfeld, N. "Musical Applications of Electric Field Sensing". In *Computer Music Journal* 21(2) Summer, pp. 69-89. 1997.

Paradiso, J., Hsiao, K., Benbasat, A., and Teegarden, Z. "Design and Implementation of Expressive Footwear." *IBM Systems Journal*, Volume 39, Nos. 3 & 4, October 2000, pp. 511-529

Pavlović, V., and Rehg, J. "Impact of dynamic model learning on classification of human motion". In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 788-795, Hilton Head, SC. 2000.

Pure Data website: http://pure-data.iem.at/

Rabiner. L. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition", *Proceedings of the IEEE*, Vol. 77, No. 2, February 1989

Rovan, J., M. Wanderley, S. Dubnov, and P. Depalle. "Instrumental Gestural Mapping Strategies as Expressivity Determinants in Computer Music Performance." *Kansei, The Technology of Emotion.Proceedings of the AIMI International Workshop*, A. Camurri, ed. Genoa: Associazione di Informatica Musicale Italiana, pp. 68-73. 1997

Sachs, Curt *Reallexicon der Musikinstrumente* (1913)

Saitek P2000 Manual: http://www.saitek.com/manuals/p2000.pdf

Sakoe, H. and Chiba, S. "Dynamic programming algorithm optimization for spoken word recognition." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-26, 43-49. 1978.

Santa Cruz Guitar Company. http://www.santacruzguitar.com/scgc_woods.htm

Sarlo, Joseph. http://crca.ucsd.edu/~jsarlo/pd/

Selker. T. Gesture Ball project webpage (2000), online at: http://cac.media.mit.edu:8080/contextweb/project?name=Gesture_Ball

Schoner, B. Cooper, C. and Gershenfeld, N. "Cluster-Weighted Sampling for Synthesis and Cross-Synthesis of Violin Family Instruments," In *Journal of New Music Research.* 1999.

Smith, Joshua. *Electric Field Imaging*. PhD Thesis, MIT Media Lab. 1999

Stack webpage, online at: http://www.media.mit.edu/resenv/Stack/

Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA. 1998.

Szilas, N., and Cadoz, C. "Physical models that learn". In the proceedings of the *International Conference on Computer Music*. Tokyo. 1993.

Therrien, C. *Decision Estimation and Classification: An Introduction to Pattern Recognition and Related Topics*. John Wiley and Sons, New York, NY. 1989

Wessel, David. "Instruments That Learn, Refined Controllers, Source Model Loudspeakers". In *Computer Music Journal*, Volume 15, Number 4. Winter 1991

Wessel, D., and Wright, M. "Problems and Prospects for Intimate Musical Control of Computers," *New Interfaces for Musical Expression* proceedings. Seattle, 2001.

Widrow, B., and Stearns, S.D. *Adaptive Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

Wilson, Andrew D., "Adaptive Models for the Recognition of Human Gesture" PhD Thesis. Massachusetts Institute of Technology. 2000

Winkler, T. 1995. "Making Motion Musical: Gestural Mapping Strategies for Interactive Computer Music." In *Proceedings of the 1995 International Computer Music Conference. San Francisco,* International Computer Music Association, pp. 261-264.

Zhang, et al, "Oldest playable musical instruments found at Jiahu early Neolithic site in China". *Nature* (401): 366 – 368, 23 September 1999