Master of Science Thesis in Art and Technology

robotcowboy: A One-Man Band Musical Cyborg

Daniel Wilcox
Göteborg, Sweden 2007

IT University of Göteborg
CHALMERS | GÖTEBORGS UNIVERSITET

REPORT NO. 2007:70

robotcowboy: A One-Man Band Musical Cyborg

DANIEL WILCOX

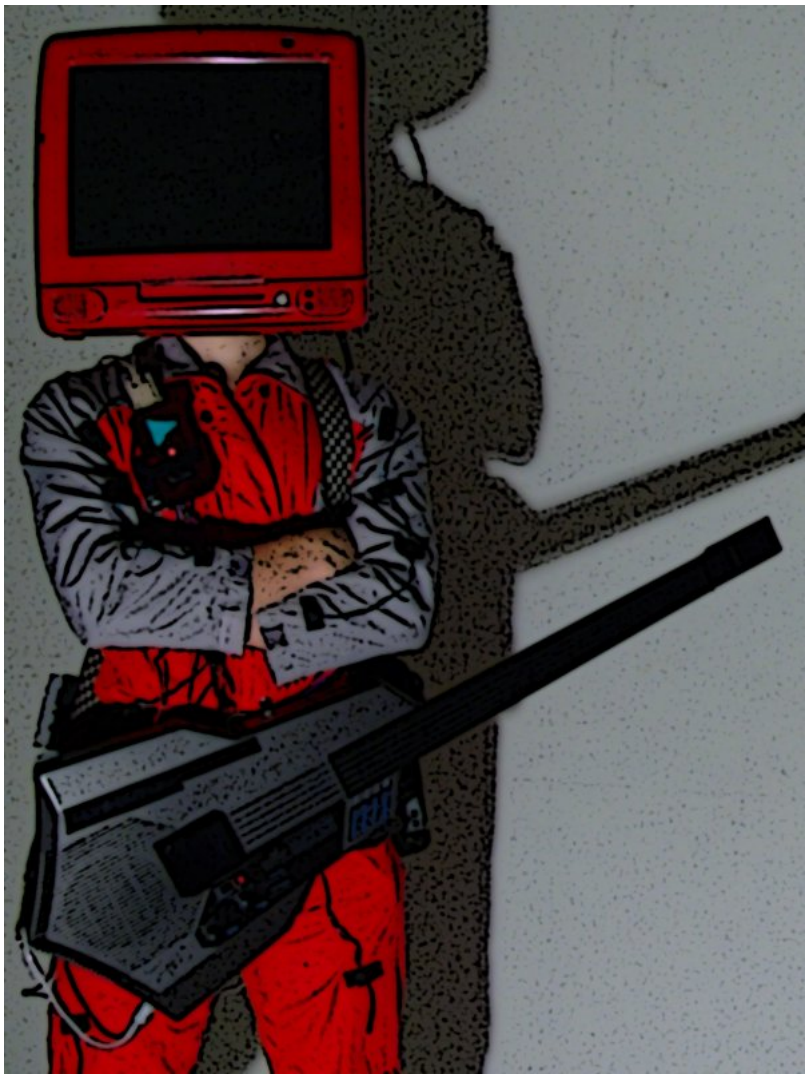
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY AND CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

robotcowboy: A One-Man Band Musical Cyborg
DANIEL WILCOX

Report No 2007:70
ISSN: 1651-4769
IT University of Göteborg
Göteborg University and Chalmers University of Technology
P O Box 8718
SE — 402 75 Göteborg
Sweden
Telephone + 46 (0)31-772 4895

Department of Applied Information Technology
Göteborg, Sweden 2007

robotcowboy:
A One-Man Band Musical Cyborg



Dan Wilcox
danomatika@gmail.com
www.robotcowboy.com

"Why, he's a reg'lar musicker!" said Button-Bright.

"What's a musicker?" asked Dorothy.

"Him!" said the boy.

Hearing this, the fat man sat up a little stiffer than before, as if he had received a compliment, and still came the sounds:

> *Tiddle-widdle-iddle, oom pom-pom,*
> *Oom pom-pom, oom—*

"Stop it!" cried the shaggy man, earnestly. "Stop that dreadful noise."

The fat man looked at him sadly and began his reply. When he spoke the music changed and the words seemed to accompany the notes. He said — or rather sang:

> *It isn't a noise that you hear,*
> *But Music, harmonic and clear.*
> *My breath makes me play*
> *Like an organ, all day—*
> *That bass note is in my left ear.*

Frank Baum, *The Road to Oz* (1909)

Abstract

robotcowboy is a performance project consisting of a wearable computer system and various peripheral devices which enable a single performer to become a mobile, technological "one-man band" free to roam the stage, the street, and the world. It is both an homage to the "one-man band" tradition and an exploration into a post-digital renewal of embodiment and physical instrumentality in electronic musical instruments. The system is built with low-cost in mind and utilizes readily-available hardware and free, open source software in order to make the concept feasible to the everyday computer performer who wishes to step out from behind his screen. It is hoped that the concept of "wearable music computer" can one day become as ubiquitous as that of "laptop musician" in a return to the fragility and excitement of live music.

Keywords: wearable computing, mobile music, performance art

Acknowledgements

This masters thesis was developed during the spring of 2007 at the Art and Technology Program, IT University in Göteborg, Sweden and the text written in June of the same year.

I would like to acknowledge and thank my supervisors Dr. Mats Nordahl and Palle Dahlstedt for their guidance throughout the undertaking of this work.

Thanks to Oscar Ramos, my partner in crime whose help and insight were invaluable during many performances where experimental equipment and software did not perform as expected and to Christian Pallin of Koloni fame for facilitating numerous opportunities to perform for the Göteborg experimental music scene.

Thanks to my friends and fellow classmates at 141:an for kitchen coffee chats, Friday dance parties, Sunday dinners (and laundry, Christine), late night soldering sessions, Tuesday basketball games, and all the rest that made our Swedish experience a memorable part of life: Salvador, Olle, Michael, Sebastian, Jeanette, Alma, Daniel, Hamlet, Eva, Christine, Fanouris, Freddy, Alejo, Anand, Pilly, Gesa, Cui Lei, Li Jing, Pong Pan, Yoshi, and the whole gang.

Thanks to Joakim for keeping the building together as it fell apart in slow motion and for making time-lapse videos of the process in action.

... and thank you DEVO for the truth about de-evolution ...

Last and first of all, thanks to Smith and Jackalyn Wilcox, my parents, for their generous support for my eccentric undertakings in a land far from Huntsville, Alabama. Mom and Dad, you may not understand what I am doing or why, but you have always encouraged be to do the best that I can, and I love you for it. One could not ask for better parents.

Contents

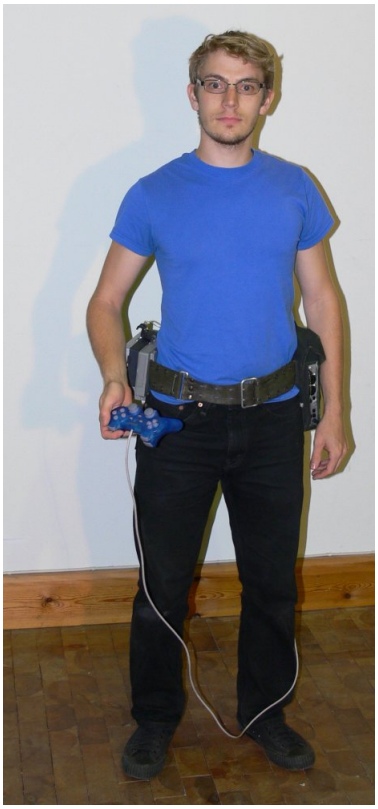


Fig. 1: The author, a musical cyborg wearing the robotcowboy system

1 Introduction

As the mobility and power of the computer continues to expand, so does its role in live music. Ever since electronic gestural controllers and synthesizers have been pioneered for the stage during the 1980's, computers and sensor technologies have expanded into the arena of live musical performance. This thesis proposes an investigation into the use of the wearable computer in this setting through the development of a human-computer mobile performance project entitled "robotcowboy". robotcowboy [sic] consists of a "one-man band" wearable computer system dubbed "unit" composed of a mobile computer and various input devices such as game and midi controllers.

The goal of such an exploration is to prompt discussion and development into the nature of the "musical performance computer". Within the last 10 years the availability of cheap yet computationally-powerful portable "laptop" computers has led to their prominence in electronic, DJ (disc-jockey),

and even pop/rock musical settings, yet their use has been criticised due, in general, to the lack of physical action, interaction, and mobility of laptop musicians on stage. Although the vast majority of electronic music being made at the time of this thesis is performed using these machines, there are notable projects and musicians who have set the stage for gestural electronic instruments that offer an embodiment of sound and instrumentality. The post-digital aesthetic of a return to the body is reemphasizing the role of the musicians movement and physical effort in the live performance arena.

The one-man band is a tradition of pragmatic creativity and independence from musical norms. The iconic man with a bass drum on his back, cymbals between his knees, drum beaters attached to his feet, and various hand and mouth instruments is at once both comical and impressive[1] — a humorous picture of a determined individual. Most traditional one-man band systems can be seen as intimate musical cyborg apparatuses, custom made for the needs of the performer, which were often taken out into the public on street corners and at musical events. This tradition continues as new technology is being adapted by performers to allow gestural and musical control beyond the limits of physical instrumentality.

The project presented in this thesis represents the struggle of one live performer applying his experience and musical requirements for intuitive expression through a technological one-man band system: the robotcowboy unit. During the course of the robotcowboy project several experimental systems were tested and the results encouraged the design goals of the final prototype. Detailed descriptions of the problems encountered and the conceptual, technical, and practical approaches to their solution are included. It is the hope of the author that those who wish to utilize the power of the computer in live music while avoiding its performative limitations will find this work useful.

[1] A tap drummer, see Figure 4

1.1 Definitions

This thesis presents several topics using specific semantics and certain words are best defined beforehand:

- **modern**: the (post-)digital age at the time of this thesis, circa June 2007; a time when Internet technology and wireless connectivity are becoming ubiquitous, mobile devices have become "do-it-all" miniature computers, and the use of laptop computers within live music has occurred for over 10 years
- **traditional**: being of before the modern era at the time of this thesis; usually used to refer to instruments and music produced using pre-digital technologies
- **post-digital**: an aesthetic created through the use of the errors and "failures" of digital technology and forcing such systems to do what they were not originally intended[2]
- **wearable computer**: a mobile computing system worn on the body; it is important to note that a stand-alone computational system, such as a modern "do-it-all" cellphone, is not considered "wearable" within this thesis as it is not integrated into the body, but designed to be carried and manipulated by the hands
- **one-man band**: a single musician playing multiple instruments at the same time; obviously, there exist "one-woman bands" and both genders are inferred for the sake of simplicity
- **cyborg**: "*cybernetic organism, the melding of the organic and the machinic, or the engineering of a union between separate organic systems.*" The Cyborg Handbook [19]
- **laptop computer**: a portable computer small enough to use on one's lap, commonly referred to as a "laptop"; at the time of this thesis, such computers are comparable in computing power to desktop computers
- **MIDI**: the **M**usican **I**nstrument **D**igital **I**nterface; since 1984, this musical protocol is the current standard protocol for controlling digital instruments through discrete events

[2] A term coined by Kim Cascone in "The Aesthetics of Failure: 'Post-Digital' Tendencies in Contemporary Computer Music" [9]

- **OSC**: **O**pen **S**ound **C**ontrol; an alternative to MIDI with greater speed, increased resolution, and flexible addressing over an Ethernet connection

- **GNU/Linux**: a free computer operating system consisting of the Linux kernel and the GNU software suite sponsored by the Free Software Foundation[3]

- **daemon**: a program that runs in the background handling continuous or periodic system functions such as power or network management

- **MAX, MAX/MSP, Pure Data**: graphical, modular, object-oriented software patching environments for realtime sound composition, processing, and generation (see also [36] and [37])

[3] http://www.fsf.org

2 Motivation

The one-man band system in this thesis was developed in order to discover new possibilities of the 'musical cyborg' — the modern combination of man and machine in order to produce live music. His tools include electronics, a computer, and computer software making him a one-man band of the new era where the determination and ingenuity of the tradition of the one-man band are combined with the power of digital technology.

As computing devices become more mobile, computer music should follow. Currently, the vast majority of live computer-generated or computer-processed music is done on stationary machines with the performer sitting or standing behind a glowing screen — the 'mobility' of the laptop is only between gigs, not on stage. This project proposes the use of wearable computers in live computer music so as to give the musical cyborg legs. The resulting 'mobile musical cyborg' can now run, jump, dance, and .... well ... rock. Laptops are 'general computing devices' and, as such, are not physically designed for active use on stage outside of sitting on a table.

In the authors experience, the vast majority of audience members at live 'laptop music' events desire some sort of interaction with or from the performer. The electric guitar, for instance is very well suited for this task in that the player can look at the audience, sing, and move about, tethered by a single cord. On the other hand, the general purpose computing device is poorly suited for this interaction since the performer rarely looks up from their glowing screen and they are physically incapable of moving from behind it. This thesis project proposes a screen-less, wire-less wearable computer performer who is free to move about the stage, whose control of sound is more tangible, and who is able to enjoy the spectacle that is live performance.

The performance art itself in the art of live music should not be subjugated by the interface of a mass-produced commercial item. The traditional one-man band strives to create a unique experience for both performer and audience by modifying readymades and custom-building instruments. Why should the one-man band musical cyborg not do the same and "realise a unique and playful thought"[4]?

[4]See the quote beginning Section 3.1

2.1 A Critical Survey on the Nature of Post-Digital Instruments

> *Many new instruments are being invented. Too little striking music is being made with them.* Sergi Jordá [25]

In order to place this thesis and its motivation within a critical and theoretical framework, it is important to review a set of critical texts on the nature of electronic instruments in the current "post-digital" era. Collectively, they present a context in which to place works mentioned within the Background in Section 3, offer a critical analysis of past and current live instruments, and argue towards a definition of the characterstics needed for an electronic instrument to be successful in a live environment.

2.1.1 The Post-Digital Aesthetic

Kim Cascone's *The Aesthetics of Failure: 'Post-Digital' Tendencies in Contemporary Computer Music* [9] begins with a quote by noted computer scientist Nicholas Negroponte: "The digital revolution is over" (1998). Cascone coins the term "post-digital" to refer to an aesthetic in a world where digital technology has lost its utopian promise through becoming commonplace. Its "failures" are no longer ignored but highlighted and manipulated through sampling errors, digital distortion, the glitch — new background sounds ala Luigi Russolo, musique concrète, and John Cage. Computers have become the primary tools for electronic music which is no longer valued for being merely "digital", it's message shifted from the medium to the tools of creation.

2.1.2 32kg: Performance Systems for a Post-Digital Age

In *32kg: Performance Systems for a Post-Digital Age* [40], Richards' notes the post-digital frustration with the black and white logic of 1 and 0 in a world of human experience in constant transition. There is a return to analog for it's "softer" nature and infinite subtlety, a renewed interest in the soldering iron and hardware hacking in the likes of David Tudor and Gordon Mumma to explore the content of the device through its native interaction. This process

of "composing within the electronics" is seen in software tools such as the patching and modular abstraction of Max/MSP and Pure Data. As opposed to the antiseptic newness of the consumer-bought digital, the post-digital is an aesthetic of adaptation and reuse where Erikki Huhtamo's "familiar aliens" [21], popular machines and symbols that have become part of cultural heritage, appear in music in the form of sampling and chip-tunes. It is a return to the personalized and self-made instrument through "punktronics" — a backlash against the "iPod future" and commercial ready-mades:

> *These are electronic instruments and working methods that are directly opposed to those of a mass produced digital culture and may include some of the following characteristics: designer trash (deliberately made to look beaten-up or broken), ugly, cheap, heavy, hand-made, designed to be handled or to come in contact with the body, ready-mades, hacked, bent, feedback and kitsch.* [40]

Richards further argues that the back-lash of the post-digital is not limited to the visual aesthetic of musical instruments but also to their interface design. Many digital input devices which are used for instrumental control are only able to accept a fraction of the gestural range of the human body. Unsatisfied by the micro-gestures of the laptop touchpad, post-digital instrument makers seek to provide interfaces more suitable to human gesure through the use of large knobs and sliders, body contacts, and exposed wires. It is a return to ergonomics and the biological reality of the body.

2.1.3 Some Remarks on Musical Instrument Design at STEIM

Despite the disembodiment of most current digital instruments, historically, there has been an interest in the body in digital instrument design. STEIM, the Studio for Electro-Instrumental Music in Amsterdam, is a research organization that pursued touch and embodiment in its instruments throughout a MIDI-revolution that further abstracted the musicians body. In 1991's *Some Remarks on Musical Instrument Design at STEIM* [41] Joel Ryan states that with the availability of digital musical devices in the period following the mid-1980's, the distancing of the composer though formalized musical processes shifted toward the composer/performer and a "quest for immediacy in music". This immediacy is almost effortlessly achieved though digital

instruments, yet it is this virtue of effortlessness, the promise of the 'digital-myth', that works against the actions involved in playing a traditional instrument. Ryan notes that physical effort is one of the functional requirements of traditional instruments which were developed and expanded with musical possibility, not required effort, in mind. In fact the progression of physical instruments moves toward an increase of required effort and thus it can be said that effort is closely arranged with expression in that more practice and muscular ability are required to draw out these expanded possibilities. Ryan argues that this inherent design principle is important for digital instruments in that the required physical actions and effort of the performance interface help bring to life the underlying musical processes, much like the physical sounding bodies of traditional instruments:

> *In fact the physicality of the performance interface helps give definition to the modeling process itself. The physical relation to a model stimulates the imagination and enables the elaboration of the model using spatial and physical metaphors. The image with which the artist works to realize his or her idea is no longer a phantom, it can be touched, navigated and negotiated with.* [41]

2.1.4 Making Motion Musical

The 1995 *Making Motion Musical* [45] by Todd Winkler discusses design principles for digital instruments which transform physical motion into sound. This is an important compositional problem since there is already a natural precedent for gestural mappings and their associated musical content within traditional instruments — slight finger tapping on a cymbal, a dramatic foot kick of the bass drum, a rolling piano arpeggio. Winkler observes that the physical constraints of both instrument and performer define the instrument's usage:

> *Physical constraints produce unique timberal characteristics, and suggest musical material that will be idiomatic or appropriate for a particular instrument's playing technique. These reflect the weight, force, pressure, speed, and range used to produce sound. In turn, the sound reflects, in some way, the effort or energy used to create it. The fact that brass tones add upper partials as they grow louder is a classic example.* [45]

These associations are what come to mind when one thinks of what makes an instrument an *instrument* and Winkler points out that electronic composers now have the ability to map any gesture to any musical parameter. This fact is a double-edged sword in that with freedom comes a greater challenge in defining meaningful and useful mappings in order to produce a more effective controller — creating an instrument. Opposite relations can be used to great effect, for instance, but care must taken not 'overdo it' as simplistic mappings can be too trivial for extensive usage.

2.1.5 The Art of Interaction

Salz's 1997 *The Art of Interaction* [42] places interactive computer works within the performance art framework: they are transient conceptual constructs, not static works and are performed either for or by an audience. An interactive work is defined as an artwork that accepts action from a human interactor and produces some sort of real world stimulus for the audience — action results in response. Salz forwards that, like dance and musical performance, these works are performing arts in which the artist builds a blueprint for the performance experience using electronics and software, becoming the composer, playwright, director, and performer in one. Live interaction and performance is a movement away from post-modern recording/repetition and towards immediacy. The medium of the performing arts is performance itself — the act of performer performing for an audience where even the spectators themselves play the roles of "spectator" and these live actions are, as Goldberg defines performance art, "live art by artists" [18]:

> *Performance is the medium. The live performance of actions is the stuff out of which the art is made. The audience regards the performance as an aesthetic object in its own right.* [42]

Salz also notes several pertinent aspects of interactive works. Performance interactions, that is the interactions of the performance defined by the mapping of the actions to responses, are broken into two types: stage interactions that are performed *for* an audience and participatory interactions in which the audience interacts directly *with* the work. In either case, a contextual environment is created around the actions of performance in that the type of performance is not defined since many works do not sit within a specific

area of the performing arts, but freely combine music, dance, and live visual arts. An interactive computer work such a realtime algorithmic musical accompaniment system can become mimetic when the the interactor, either performer or audience, personifies the system as an interactor itself. In this case the system becomes an instrument separate from the other performers whereas in more participatory interactions the interactive element is integral to the performance.

2.1.6 Digital Instruments and Players: Part I — Efficiency and Apprenticeship

Sergi Jordá, co-creator of *Afasia* (see Section 3.1.3), critiques the status of new musical instruments in his 2004 *Digital Instruments and Players: Part I Efficiency and Apprenticeship* [25] stating that there exist few studies for the design of such instruments — tools for playing and making music as a conceptual whole. No recent electronic instrument has attained even the small popularity of the Theremin and Ondes Martenot between the 1920's and 50's and, in fact, the latest new instrument to create its own compelling sound, music, culture, and virtuosi is not even digital or electronic: the turntable of the 1980's. He lists his dissatisfaction with the status of electronic instruments as a lack of dedicated performers and instruments:

> *New instruments possibilities are endless. Anything can be done and many experiments are being carried out. Yet, current situation and results can hardly be considered awesome.*
>
> - *The list of new instrument virtuosi and/or professional musicians who use them as their main instrument is surprisingly small (Michel Waisvisz, Laetitia Sonami, Nicolas Collins, Atau Tanaka, Richard Boulanger).*
> - *Being that live electronics and laptop music is so widespread [9] it is symptomatic and frustrating that so many performers prefer to still rely on the mouse, or at the most, on generic and dull midi fader boxes.*
> - *Commercially available new instruments are scarce and hardly imaginative and ground-breaking (e.g. Korg KAOSS Pad).*
> - *A new standard electronic instrument is yet to arrive.*

Jordá further states that in the last 2 decades, many new musical controllers have been developed that enable virtuosity and can interest novice users, yet have not encouraged deeper exploration and creativity. For a new instrument to be successful, it should be properly balanced to encourage use by both professionals and novices. A simple instrument is easy to pick, but short on musical possibility while a complex one can be too intimidating to the beginner, alienating the user before its subtleties are explored. The piano, for example, is easy to play while offering enough possibility for long study and mastery. A good instrument should not impose its music on the player, it should not only play "good" music but must have a capacity for failure, for the limitations of "bad" music offer musical freedom of choice.

2.1.7 Theses on liveness

John Croft's 2007 *Theses on liveness* [10], although focusing on a more academic combination of classical and electronic instruments, offers a critical analysis on the nature of current and future digital instrument designs. He states that most musical projects thus far have placed an emphasis on the *technical* relation between performer and computer, but have trivialized the *poetic* relation of musician and instrument by a dislocation of the sound and source in electronic music. This may stem from an adherence to one viewpoint in the division of two opposing sides in late twentieth century music: an attempt to reemphasize the body in performance versus the disembodiment of sound for aesthetic freedom. The latter attempts to remove the body as a means of obvious production through musical gesture without physical gesture — an end to the age of the concert, its associated "spectacle", and the necessary evil of the live musician. Striving to free listener and sound from presupposed relationships, this music can be referred to as "acousmatic", a term which is derived from Pythagoras's method of lecturing from behind a screen so as to focus the minds of the audience on his words. Many artists in this tradition believe the ideal setting for a listener is a quiet room and a pair of headphones so, naturally, their is a disconnect when acousmatic music is performed live:

> *This acousmatic character is often cited as one of the difficulties with the reception of acousmatic music — not, it has to be said, so much because it erases the labour of production, but more often because there is nothing to look at. Thus there have*

> *been various attempts to reintroduce the visual, from video projections to a focus on the person behind the mixing console as diffusion artist. The former addresses the perceived need to accompany sound with images, without attempting to address the aforementioned de-corporealisation. The latter, in contrast, is borne of the desire to re-incorporate human performance, but it encounters a familiar problem: while there is a body, there is only a generalised mapping of the physical movements of such a body (pressing keys, moving faders, and so on) to the types of energy and gesture present in the music the music remains, in essence, acousmatic, in the sense that what is known to be the source is visible but remains perceptually detached.*

Croft discusses "liveness" and breaks it down into two types: procedural and aesthetic. Procedural liveness is the plain fact that input and input sound is transformed to output sound in real time and aesthetic liveness refers to aesthetically meaningful transformations between input and output sound that is achieved through procedural liveness. As noted previously by Salz and Winkler, aesthetic liveness relies on the link between the performers action and the computers response in live electronic and computer music. If the musical mappings are too transparent, one-to-one, then the result can be banal since it becomes a technology demonstration that highlights the procedural processes. If many actions yield no appreciable differences in the sound, as in most laptop and knob/fader driven performances, the liveness, once again, is perceived as being only procedural — thus the medium becomes the message. An appreciable delay between action and result renders the liveness procedural as well and, in this case, the performance might as well be prerecorded. As discussed by previous authors in this survey, the instrumental context of physical action and resulting sound is important in making a device an *instrument*:

> *... simultaneity is also closely linked to two fundamental principles of live performance: first, we expect a meaningful relationship between what we see the performer do and the sound that this action generates; second, as Simon Emmerson*[5] *points out, [w]e expect a type of behaviour from an instrument that relates to its size, shape, and known performance practice.* [10]

[5]Emmerson (1998: 148) [14]

The laptop musician, for example, ignores the first principle in that the performers gestures are opaque since the audience expects a sound in proportion to the energy of the performers action. It can be argued, however, that this opacity of relation can be present for an audience unfamiliar with the physical mechanics or cultural notions of virtuosity particular to a traditional instrument. This lack of understanding, however, is mediated by a physical interface whose mappings are still more obvious then that a typical electronic controller. Even to a completely unfamiliar viewer, traditional instruments are more readily explored beyond cultural stereotypes down to the essence of action and result, that the limitations of the instrument, once again noted by Ryan and Winkler, are what make the performance compelling:

> *This is surely why performance engages us in a way that cannot be accounted for in terms of the sound alone: the difficulty, the impossibilities, the encounter with limits, the finitude of the instrumental performance resonates with wider human experience. This dimension of instrumentality is precisely what needs to be understood if live electronic performance is to mean anything beyond the trivial fact of someone pushing buttons while we listen.* [10]

In order to lay down a blueprint towards electronic instrumentality, Croft posits a set of conditions for an effective instrumental relationship which include the relationship between action and response as well as its creative depth. Action and response must be proportional, a small action yields a small sound whereas large movements bring about large changes of response and action. A morphological connection must exist between the type of gesture and musical response: bowing sustains a musical process while plucking yields a more staccato result. As forwarded previously, these relations must be synchronous and consistent in order to preserve the connection between performer and sound. Croft, as Jordá before him, requires that the instrument be both learnable by novices and sufficiently deep and fine-grained for long-term use and expressiveness. These requirements are built on the recognition that the fact a computer can generate sound in response to an action is uninteresting in it own right and that the triggering of live sound using sensors is often dull or merely interesting — that there is more in a live performance:

> *The problem, then, for any live electronic music that would re-*

> *alise the instrumental paradigm, is to address not only the gestural, morphological and spatial disjunction in purely aural terms, but somehow to create the unified expressive persona normally associated with a solo performance, which is so easily destroyed by the rigidity and disembodiment of the electroacoustic sound.* [10]

Finally, Croft observes that the *grain* or *genotext* of performance and the fragility of the act itself is uncodifiable and unprogrammable. The fact that the performance can fall apart at any moment is what makes live musical performance exciting: the tightwire, the high speed car race, the blast-off of a manned space vehicle all balance upon the precipice — an act of orbiting, as in falling towards the earth and missing. By using "perfect" live electronic instruments, immutable recordings, and opaque software processes, the inherent limitations of both performer and instrument are essentially eliminated and the performance subliminally lessened by the exclusion of the excitement of failure. It is the quest toward perfection that characterizes recorded music, but for a live electronic instrument to become successful, it must, as Jordá emphasizes, have a capacity for failure — for "bad music". If the "spectacle" of live music is the tightrope act between failure and success, then this author, for one, is willing to take that risk.

3 Background

The development of this thesis project was inspired by the tradition of the "one-man band" and numerous previous performance projects. Examples of traditional and modern one-man bands are presented and a relevant discussion of this tradition as a "cyborg-entity" is examined. Notable previous performance projects are highlighted which introduce digital instruments and controllers, integrate mobile/wearable computing and the body, focus on ways in which to make the computational generation of sound tangible.

3.1 The One-Man Band

> *"There is something deeper at work in this extraordinary impulse to play it all, alone, at one time, with all the requisite physical agility, and to play it so joyfully. There is a radical independence at work here, an urge to confront and explore human capabilities and possibilities, an urge to realise a unique and playful thought."*
> Hal Rammel [38]

As defined by Hal Rammel, whose 1990 *Joe Barrick's One-man Band: A History of the Piatarbajo and Other One-man Bands*[6] provides much of this background, a "one-man band" is a single musician playing multiple instruments at the same time; — an ensemble limited only by the dexterity and ingenuity of it originator. This tradition, although often attributed a novelty status, has a well-defined historical record which spans cultural and political boundaries. Of interest is the persistent drive of some musicians to "go it alone" for various reasons and the solutions they come to through a true melding of man and technology. It can be said that the one-man band has been a cyborg hidden in plain site. Man's historical use of technology as an extension to musical abilities is more and more relevant as the use of computer technology in live music becomes a staple of modern times. The one-man band of yesterday may not recognize the version of today, but both share the same passion to realize their music.

[6]The next entire section, Traditional One-Man Band Instruments, consists of quotations and paraphrasing from this article [38]. Those interested in more detail are highly encouraged to refer to this source!

Fig. 2: Elizabethan clown Richard Tarlton playing the pipe and tabor, 1400's

3.1.1 Traditional One-Man Band Instruments

History contains a collection of one-man band instruments that have survived, in one way or another, into cultural tradition.

The oldest noted combination of instruments by a single performer are the pipe and tabor, first referenced in the 13th century. The pipe is a small flute played with the left hand and the tabor a drum suspended from the wrist or shoulder which is beaten by a stick in the right hand. It was commonly used by traveling minstrels and clowns in Medieval Europe as shown by a 15th century woodcut of an Elizabethan clown, Richard Tarlton (see Figure 2). The pipe and tabor can still be heard in rural areas of France and Catalonia as well as in some South American native music.

The stick zither and "stump fiddle" are single-stringed "rhythm sticks" consisting of a stick set with a resonator. The simplest version, found throughout Europe and referred to in 11th century Nordic sagas, is strung with a pig's bladder at the lower end and played by scraping a notched stick across the string producing a percussive sound not unlike a snare drum. When a suitable noisemaker or bell was attached and the entire apparatus stamped on the ground, the instrument could function as a full rhythm section in accompaniment to carnival bands and was even sold commercially for this purpose in 1890's Germany. The American stump fiddle is a similar instrument used for comic effect in Vaudeville with the bladder replaced by

Fig. 3: Example of a modern "Stumpf Fiddle", 2006

Fig. 4: A tap drummer and his kit, 1960's

a tambourine, box, or even items such as a washboard. Today's variant, the "Stumpf Fiddle" manufactured by The Fiddle Factory of Sheboygan, Wisconsin, Figure 3 consists of a rubber-footed pole strung with a large coiled spring over a pie tin filled with BB's and is adorned with a horn and bicycle bell.

The tap-drummer's kit, Figure 4, can be seen as the evolution of the rhythm stick onto the body in which the performer becomes the stick whose gestures are mapped directly to sound. It generally contains an assortment of drums and cymbals worn on the back and shoulders operated by the feet. By far the most widely known symbol (and stereotype), this one-man band setup provides percussion accompaniment through the simple act of walking while other instruments are played with the hands and mouth. The tap drummer's cymbal developed into the "sock-cymbal" and later into the modern hi-hat. In fact, the hi-hat stand, the bass-drum kick pedal, and the entire modern drum kit can be attributed to big-band drummer Gene Krupa who helped establish the concept of a single drummer as entire rhythm ensemble in modern music.

Much in the essence of the tap-drummer, the well-known harmonica rack allows a performer to play a harmonica or kazoo while leaving the hands free to play other instruments such as drums or a banjo. Originally available commercially in early Twentieth Century America, a rack was often fashioned by hand by those who were not aware of the fabricated version. Numerous travelling musicians utilized this device to entertain at work camps, festivals, and Vaudeville. As such, it became ingrained in the American folk tradition and the combination of guitar and harmonica has become so commonplace that it is not even associated with the one-man band.

3.1.2 Unique Traditional One-Man Bands

Numerous examples exist of unique and novel approaches by traditional musicians seeking their own accompaniment. Examples include Vaudeville artists, Jesse Fuller, Fate Norris, Joe Barrick, and Rahsaan Roland Kirk.

One-man stage acts routinely combined piano, guitar, vocal, and percussion sequences and Rammel details several unforgettable performances of the Vaudeville and music hall venues in the early 20th century:

Fig. 5: Jesse Fuller, 1950's

Ragtime composer Wilbur Sweatman in the early 1900s did a vaudeville act playing three clarinets at once and Vick Hyde, a vaudevillian of the 1940s did his finale playing three trumpets at the same time and twirling a baton as he exited the stage. Virtuoso Violinsky concluded his act with a piano-cello duet by fastening a bow to his right knee while his right hand fingered the strings, leaving his left hand to accompany himself on the piano. The piano, generally thought to be a two-handed instrument was played with only the right hand by Paul Seminole in the 1920s while he played guitar with his left, and for jazz musician and comedian Slim Gaillard playing the piano and guitar at the same time was possible by turning up the volume on his electric guitar "... it'll play itself - you just make the chords and hit the strings, feedback!" [38]

Jesse Fuller, one of the most widely known and recorded one-man bands, invented several devices including a foot-operated bass in order to expand his music. Born in Georgia in 1896, Fuller, Figure 5, did not start playing professionally until the 1950's when, after a life of hoboing, crafting, and World War II ship-welding he decided to turn to music. He had learned to

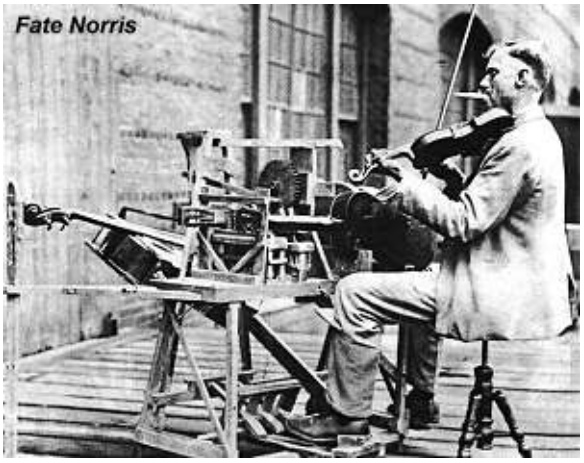


Fig. 6: Fate Norris, late 1920's or early 30's

play guitar at an early age and developed his own accompaniment as a one-man band: the 'footdella', an upright bass whose 6 strings were struck by individual foot-operated hammers, a sock-cymbal, and a special head rack to hold a harmonica, kazoo, and microphone. His huge repertoire of music ensured a good working schedule with performances in the US and Europe and his well-integrated playing brought numerous recording deals.

Bluegrass fiddler Fate Norris toured an entire one-man string band at fairs and fiddling contests throughout the Southern United States in the 1920's and 30's. Best known for playing banjo in the hillbilly string band The Skillet Lickers, Norris constructed an elaborate arrangement of guitars which he controlled via foot pedals while playing a fiddle and kazoo (Figure 6). No recordings exist of the performance which was described in a newspaper article when the Skillet Lickers appeared in Nashville, Tennessee in 1927:

> *Fate Norris, of Dalton, Georgia, the one-man wonder, who plays six individual instruments in an individual band, will also furnish entertainment. Mr Norris has in his band two guitars, bells, bass fiddle, fiddle, and mouth harp. He devoted seventeen years to mastery of his art.[46]*

Joe Barrick, a carpenter and musician, turned to his skills and ingenuity when it became hard to "keep anybody together to play with anybody" [38]. Born to Native American Choctaw parents in Oklahoma in 1922, he began

Fig. 7: Joe Barrick and his piatarbajo, 1980's

playing the mandolin at age 15 and quickly absorbed music off of the radio. After serving in the armed forces and settling in California as a carpenter, Barrick began playing in groups and building his own instruments — first being a guitar built out of a cows skull. When it became hard to keep a full band together, he decided to perform as a solo act and approached the problem of playing rhythm guitar using his feet.

The result was the "piatarbajo", a shelf arrangement containing a board-mounted guitar, bass guitar, banjo, and snare drum (see Figure 7). The right foot controls hammers that strike the instruments and the left works special treadles which operate move-able frets that the chords to be played by pushing on the appropriate strings of the guitar, bass guitar, and banjo. Each instrument has a separate pick-up played through a separate amplifiers — a bass amp for the bass guitar and a corresponding guitar amp for the guitar — and he would arrange the speakers so that the entire performance sounded like individual musicians were playing. Motivated by music as a social event, he toured schools, festivals, and dances and his creative independence was evident:

> *No one tells me when to practice and I can play any song I want without having to hope the rest of the band likes it.* [38]

Jazz musician Rahsaan Roland Kirk, Figure 8, sought to recreate the

Fig. 8: Rhasaan Roland Kirk, 1970's

sounds of his dreams and became his own one-man band as a result. Although most one-man bands are thought of as mere novelty, Kirk's 'serious' approach to the tradition enabled him to achieve a true innovation. During live performances he hung drums, flutes, and whistles around his neck and arrayed gongs, a sock cymbal, and a bass drum at his feet. He developed a method of playing 3 saxophones at once and could play 2 entirely different melodies while improvising a third and playing rhythm with his feet simultaneously. Roland Kirk's sheer physical ability and novel approach to achieve his goals resulted in a unique musical expression.

3.1.3 Modern One-Man Bands

The one-man band is a rich musical tradition that has been carried on and adapted with the evolution of technology. Examples include the pioneering work of Michel Waisvisz, the elaborate electromechanical constructions of the Japanese performance group Maywa Denki, the self-titled "One-man multimedia band" Afasia, and the ubiquitous laptop musician.

Dutch composer and performer Michel Waisviz has developed novel electric and electronic instruments in order to play his pieces. Largely self-taught, he focuses on touch and the act of making sound tangible and directs the Studio for Electro-Instrumental Music (STEIM[7]) in Amsterdam [29]. Waisviz is credited with developing one of the first digital gestural music controllers, *The*

[7]http://www.steim.org/

Fig. 9: *The Hands, late 1980's*, Michel Waisviz

Hands (see Figure 9), which consists of wooden shapes fitted for each hand with buttons for each finger, tilt sensors, and range sensors between each device. Using *The Hands* and *LiSa*, live performance software co-developed at STEIM, he is able to masterfully control multiple sounds and musical events in motions reminiscent of the space-controlled Theremin. [20]

The Japanese performance group Maywa Denki creates elaborate musical devices controlled by single performers. Best known for their device art "Bit Man" and "Knock Men", Maywa Denki produces mechanical guitar, flute, and percussion players for its show *Tsukuba Series* in order to "stir people's attention to the fact that the live musical sound is created from a substance (musical instrument)" as opposed to "information" stored on a sampler, synthesizer, or personal computer. Performers utilize *Ton-Ton Kun* switch pad controllers to control several electromechanical instruments simultaneously. [12]

Developed by a collaboration between visual artist and performer Marcel.lí Antúnez, mechanical sculptor Roland Olbeter, and engineer Sergi Jordá[8], the one-man digital theater play *Afasia* allows its single performer to conduct

[8]See section 2.1.6 for a paper by Jordá

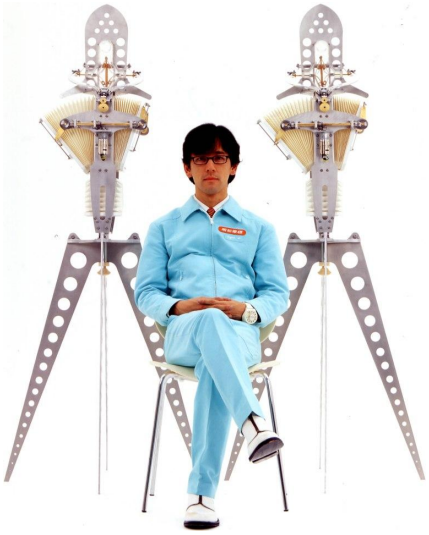


Fig. 10: Maywa Denki Pres. Nobumichi Tosa and 2 Mechanical Instruments, early 2000's

Fig. 11: Marcel.lí Antúnez and his sensor exoskeleton in *Afasia*, 1998

music and control animations and video. Antúnez's custom sensor exoskeleton, Figure 11, features keyboard-like switches mounted on the chest plate and tilt-sensors placed on the fingers which wirelessly feed data parameters to a central computer. In turn, this computer controls computer generated animations, DVD video, and a robotic quartet consisting of an electric guitar, one-string violin, a drum, and a three-bagpipe "horn-section". Basic gestures are mapped to a set of control commands including the muting, playing, looping, and transposing of sequenced musical tracks. A 'solo-mode' also allows for direct control of the robotic instruments:

> *While the previous commands use simple one-to-one mappings, solo modes apply more sophisticated gesture detection mechanisms, that allow for example to play the robot-guitar in an almost conventional manner, controlling pitch with the performers left elbow angle, while triggering notes with different right-arm fast movements that enable the performer to play chords, arpeggios or monophonic lines. Twelve solo modes (three for each robot) have been defined.* [24]

Since the late 1990's, the laptop musician has become one of the most ubiquitous and powerful one-man bands. Today's machines can easily process realtime audio, handle a huge assortment of sound files, and be used as live improvisational instruments. Much like the commercial rhythm sticks of yesteryear (see 3.1.1), the laptop can be thought of as a Duchampian "ready-made" which can be adorned by all manner of acoustic, electric, or digital noisemakers [4]. The ingenuity of the performer is apparent through the nearly limitless sonic combinations available through commercial or custom software, modified or "hacked" electronic devices, and live sound processing.

It is important to note that, although the laptop musician has become more and more common, the general cultural reception of live "laptop music" is still largely cold: "to play live with turntables and a drum machine goes unchallenged, but for a laptop producer to appear alone behind a laptop without visuals is still resisted by many". One reason for this reaction is that the computer itself stands as a symbol for de-humanisation and the complete abstraction of sound into raw data somehow removes this essential 'human element'. [32] A second reason is that many onlookers fail to see the performative aspects of laptop music and will become confused, lost or bored due to the perceived loss of visual, aural, and tactile interaction

between the performer and their instrument[9]. Audience members outside of the experimental and electronic music scenes have not yet accepted the performance of laptop musicians much as many of a previous generation rejected synthesizers which now have become so commonplace that their use is no longer questioned. Listeners, instead, are asked to focus on the "aural performativity" as opposed to visual and physical actions in order to fully appreciate the art of the laptop musician. [43]

3.1.4 Relationship Between Instrument and Musician

The one-man band can be thought of as the ultimate combination of performer and instrument maker in that most performance systems are custom made. In *The Haptic Sensation and Instrumental Transgression* [39], Pedro Rebelo states that a relationship exists beyond the simple act of the production of sound in that each instrument defines ways to be touched, felt, and activated. In order to explore the instrumental relationship, Rebelo further proposes thinking of the instrument, not as a tool to be controlled, but as a multimodal participatory space in which it is as much an entity as its player. The performer participates in a 'Smooth Space', as identified by Deleuze and Guattari [11], which is navigated by a constant reference feedback loop between player and instrument.

This feedback loop is integral to the one-man-band since most performers construct their own instruments, in effect creating musical partners. The notion of the instrument as an 'entity' that carries its own context and distinction is largely absent in Western music tradition, whose prevailing trend is towards the standardization of instruments, tuning systems, and timbre. In the case of African musical culture, however, instruments contain a unique context because they are built based on need, availability of construction materials, and the experience of the builder. It is within this context that a melding of African and Western musical traditions occurs within the one-man band. The construction and performance of these instruments and systems can be seen as the birth and training of new musical entities in the same vein of the African instrumental tradition:

> *[E]ach musician makes his own instrument to suit his own particular tastes. He also teaches the instrument the language it will*

[9]Refer to the texts by Jordá and Croft in Section 2.1 for more detail.

speak which is, of course, the musicians own mother tongue. [5]

3.1.5 The One-man Band As A Cyborg Entity

> *Cyborg. Cybernetic organism. The melding of the organic and the machinic, or the engineering of a union between separate organic systems.* The Cyborg Handbook [19]

In the tradition of the one-man band the relationship between performer and instrument is also that of man-machine — the one-man band is a "cyborg entity" with the deep relationship between performer and instrument. Strictly speaking, a "cyborg", short for "cybernetic organism", is a combination of organic flesh and mechanical and/or electronic parts, such as the pacemaker. [19] For the purpose of this discussion, the term "cyborg" will be used more loosely to refer to the man-machine combination outside of the body, as in eye glasses, which act as an extension to the body allowing the user to perform actions they could not normally carry out. Most traditional musical instruments can be seen as cyborg extensions: a woodwind instrument allows the player to produce sound using breath, mouth, and finger action — without the reed and hollow tube this would not be possible. In the same manner, the instruments and systems of the one-man band allow the performer to play multiple instruments designed for a single performer simultaneously which would otherwise not be physically possible.

In the case of custom-built instruments such that of the one-man band, the "cyborg entity" has a large performance capability compared to standardized instruments since the machine half of the cyborg fits more readily with the needs of the human half. The standardization of instruments, as noted earlier in section 3.1.4, has resulted in standard lengths, sizes, and types of instruments, yet standardization cannot fully take into account, for instance, the physical difference in size of the human hand from person to person. Generally, physically small instruments are played by physically small musicians and the same is true for large instruments and large musicians. If a "small" person wants to play a "large" instrument, the instrument can be modified or custom-built in order to better fit its player and result in a better relationship between musician and instrument. Thus in creating one's own cyborg attachments and enhancements, the builder can better realize his own goals as a musician.

3.2 Relevant Previous Works

Many notable performance projects were examined in the course of this thesis' development for their use of mobile computing, attempts to make sound tangible, and practical considerations for live performance. The "Soundwalk" is a technologically mediated exploration of the urban environment included due to its use of mobile computation. Exceptional interfaces such as the BioMuse and the Body Coder represent radical departures towards embodied sensor technology and sound. Several projects have focus on wearable musical instruments and interesting examples of tangible performances are included.

3.2.1 The Soundwalk

Sonic City and *Sonic Interface* are mobile computing projects which map environmental information into live sound in the form of the "soundwalk". Gaye, Maz, and Holmquist's *Sonic City* maps "mobility as interaction" through the everyday walk of a city dweller and generates sound using movement, light, and sound sensors. The prototype system is designed to be worn on the body utilizing a laptop computer, stereo microphone, headphones, and special sensor jacket. Much focus was focused on the sound control and design space with a framework built for context-aware sound generation. The result is a mobile, expressive, and responsive experience which highlights the movement and aspects of the users environment. [17]

Maeybayashi's *Sonic Interface* (Figure 12) is "an extension for the ears" which highlights previously unnoticed sound through realtime sampling and remixing as the user traverses their environment. The system consists of a laptop computer running the realtime audio processing software MAX/MSP, microphones to sample sounds around the user, and headphones for the playback of the processed sound. It functions as "an auditory filter which modifiers all the sound the subject encounters" and "[c]hanges in a sense of time and place caused by this equipment will make the subject conscious of exchanges between the body and the environment" [31]. The user wears the system in a backpack and is guided by a navigator for safety reasons. Reality becomes replaced by a "virtual reality space" in which participants can "see everyday things" but a feeling of reality" becomes very thin. [26]

Fig. 12: Maebayashi's *Sonic Interface* in use

Although these projects present an engaging user experience, they were not designed for direct music control which is an important requirement for this thesis project. The use of sensors mounted on the body provides input which is only controlled by the user in a passive sense, they do not have direct access to the sound processing mappings beyond the mediation of sensors and microphones. As the goal of these projects is to encourage exploration of ones environment such mediation is not an inherent failure and, in the scope of this thesis, each project's focus on mobility is relevant.

3.2.2 Exceptional Body Interfaces

Numerous digital instruments were developed during and after the MIDI revolution of the mid 1980s and early 1990s with several notable examples in *The Hands*, Body Coder, and Bio Muse. As noted earlier in section 3.1.3, Michel Waisviz developed *The Hands*, Figure 9, in order to control digital sound processes in an organic, gestural manner. Considered the first digital gestural music controller, it consists of wooden shapes fitted for each hand with buttons for each finger, tilt sensors, and range sensors between each hand. [20] The author has witnessed a live performance by Waisviz using *The Hands* and, unlike many such devices, the instrument becomes a true extension of the musician as Waisviz appears to grab and arrange "sound itself".

Bromwich and Wilson's Body Coder system is "a sensor array designed to be worn on the body of a dancer" which communicates over a custom ra-

Fig. 13: A BioMuse

dio system. Resistive bend sensors are positioned over body joints and sewn into a skin-tight costume and 4 switches mounted within a glove allow mappable function control. Sound generating devices are controlled via MIDI and control-voltage outputs from the radio base-station and several performances using the system have utilized the MAX/MSP realtime audio-processing environment. From its inception, the system is meant to be a "hyperinstrument" to "transform the dancer into a musician/instrumentalist". [7] This melding of two aesthetic art forms, dance and music, is a non-trivial endeavour and the developers of the system have acknowledged this fact:

> *In attempting to 'liberate' the dancer from his/her traditional role as subservient, by providing him/her with the means of controlling and manipulating sound, in attempting to reduce the gap between composer and choreographer, in attempting to create an interface between dance and music through the use of new technology we, like a number of our colleagues, have created another gap; the gap between the interactive and the non-interactive and the skill based difference between the single established art-forms of dance and music, and the seemingly, poor and less skillful sister, the interactive genre.* [8]

The BioMuse, Figure 13, is a musical interface which senses the biological signals of the performer. Developed at Stanford University, the system generates serial and MIDI data based on realtime electroencephalogram (EEG), electromyogram (EMG) and electrooculogram (EOG) information relayed through surface gel electrodes on the skin. [28] Atau Tanaka of SensorBand (see section 3.2.4) plays this device using the muscle movements of his arm

Fig. 14: Atau Tanaka performing with a BioMuse

and chest to drive custom patches in the MAX/MSP realtime software environment (Figure 14). For Tanaka, the electrode placement, BioMuse hardware, and computer forms the instrument and the MAX patch the musical score. The entire system can be seen as an extension of the gestural music controller in which gesture becomes the instrument itself: *"... my use of the BioMuse is an abstraction of instrumental gesture in the absence of a physical object to articulate music through corporeal gesture.* [44] Interest in the device and its applications have spawned several similar projects such as the MiniBioMuse which "costs about one hundredth [of the BioMuse], [is] small like [a] VHS-tape, [and] very light and portable with [a] battery" [33].

These body interfaces are quite complicated and are the focus of projects in their own right. The direct use of the body as a gestural instrument requires more elaborate systems than those used in this thesis, but the concept of the body as instrument itself is very relevant to physical performance. Since cost is one of the goals within this thesis project (see section 5.1), the expense of such interfaces is prohibitive and, as is such with advanced technology, their "wow" factor can cause the audience to focus on the mechanics of the devices rather than the musical and performance aesthetics.

3.2.3 Wearable Interfaces

Interfaces designed to be worn have been proposed and prototyped in previous projects such as the MIT Musical Jacket Project and CosTune. The MIT Musical Jacket (Figure 15) Project was developed in order to explore a way

Fig. 15: The MIT Musical Jacket, 2000

Fig. 16: The CosTune jacket varient, 2001

in which to "merge computing "seamlessly" into our everyday shirts, shoes, or eyeglasses". A capacitive fabric keypad sewn over the left pocket controls a microcontroller circuit which in turn signals a custom compact MIDI synthesizer. Sound is played via a miniature set of speakers or external audio amplifier. The electronics can be removed and the jacket washed normally. [35]

CosTune, "costume" + "Tune", is a set of wearable musical instruments designed for live, wireless ensemble jam sessions. Custom mobile gestural instruments including gloves, a jacket (Figure 16), and pants seamlessly connect to a server over an ad-hoc network and, as with network performance projects, this medium connects the performers. The players receive the musical control data of each other in order for their portable synthesizers to generate the performance on headphones. The main goal of the project is to facilitate communication using music — "We designed 'CosTune' to be a

communications tool rather then a simple musical instrument" — and, for example, the wireless coverage of the CosTune server is kept within 50-100m in order to make sure players can locate each other visually. [34]

For the requirements of this thesis, both the Musical Jacket Project and CosTune suffer from simplicity as they are effective interfaces, but are too basic to be useful musical expression systems. The keypad layout of the Musical Jacket allows simple note triggering but is not useful beyond its technological novelty. The gestural nature of the CosTune wearable interfaces are more flexible but, however, the musical mappings and aesthetic description of this important aspect are lacking in the documentation of the project making it hard to gauge the musical performance. As with many such interface projects, the musical output is not a main focus beyond "playing sounds".

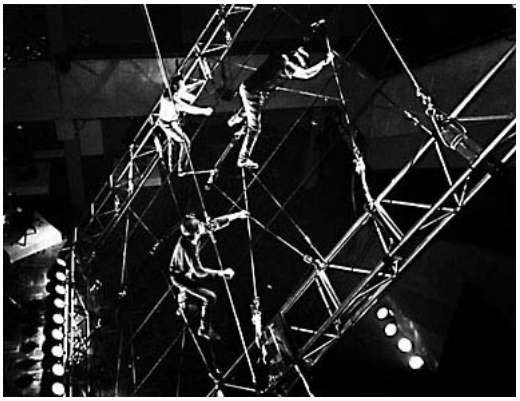


Fig. 17: The SensorBand *Soundnet*, 1990s

3.2.4 Tangible Performances

Tangible musical performances, in the realm of this thesis, are performances in which the generated sound is the result of a physical action of the performer and in this sense SensorBand, the *Afasia* performance, and the works of Laurie Anderson are relevant. The SensorBand trio of van der Heide, Zbigniew, and Tanaka performed throughout the 1990's using sensor-based gestural computer music controllers such as the *Soundnet*, a giant musical web for "human spiders"(Figure 17); the BioMuse [10], a biological signal interface; and the *MIDI-conductor*, "an ultrasound-based spatial hand-held instrument". The performance of these instruments strongly links the actions of each performer to the sound that they produce as with a traditional ensemble. SensorBand has toured and performed within both academic and commercial venues and their music is full of energy "at the roots" of tradition:

> *SensorBand's music is contemporary, non-commercial, and we use new technology. We want to communicate. Our concerts exploit energy, we want the audience to feel like they have just gotten their batteries reloaded. We want them to feel stronger and like better human beings.* [6]

As mentioned in section 3.1.3, *Afasia* is a "one-man-multimedia-band" whose single performer conducts music and control animations and video. A custom exoskeleton transmits multiple switch and tilt-sensor information to

[10]See Section 3.2.2 for more infomation on the BioMuse

Fig. 18: Laurie Anderson, 2000

a central control computer which controls video and four robotic musicians. The show is performed in a theater setting with the robot quartet occupying the front of the stage. One of *Afasia*'s design goals was for the control of the performer over the audio-visual system to be apparent to the audience. Interaction protocols were designed to be "visually obvious" and the whole system was built "according to the personality of the performer, who likes to move in brusque and coarse movements" [24].

A premier modern performance artist, Laurie Anderson's work has included numerous stage performances utilizing technology "as a means of representing the individual in the modern world of mass culture and hi-tech simulations". For *Home of the Brave*, she dances a "Drum Dance" using a custom suit with built-in drum triggers which are played with large, mechanical movements as if controlled by invisible puppet strings. Anderson's "Light Suit" is a both a costume and lighting device which illuminates and maps the performers movements within the theatrical space. For Anderson, the body becomes the "ultimate portable instrument" [3] embodying performer, props, set, and soundscape. [23]

The work of these projects as tangible performances has influenced the design of this thesis. By connecting bodily movement with the generation of sound, the bodiless technology that creates this sound becomes part of the performer. It is this embodiment and emphasis on the relation of action to sound that is most relevant to the robotcowboy project.

Fig. 19: The robotcowboy helemt

4 Experimental Performances

During the genesis of the robotcowboy project, several preliminary experiments were conducted in the realm of performance using technology: the "robotcowboy helmet" and the "robotcowboy button_box" [sic]. The robotcowboy helmet is a computer-monitor mask which turns the wearer into a face-changing icon of modern computing and the robotcowboy button_box is a wireless play button for automatic play list control of music software. These tests of interaction and performance principles helped shape the main goals, structure, and substance of the robotcowboy "unit" detailed in Section 5.

4.1 robotcowboy helmet prototype

The robotcowboy helmet, Figure 19, is a computer monitor worn on the head for human computer performance. This head-covering mask is built out of an Apple iMac computer case, lcd monitor, motorcycle helmet, and plastic foam. Since the face is completely covered, vision is supplied via a small video spy camera mounted just above the monitor and the signal is viewed through a pair of video goggles. The case was chosen due to its availability and the Apple design aesthetic which features a convenient construction for dis-assembly/reassembly, sufficient space for the various components, and a good proportionate size in relation to the authors body. A wearable computer (see Section 5.2.1) provides the images and "faces" on the screen itself and a sizable lead-acid battery worn on the waist can power the 33W display for almost 2 hours.

As a tool for performance, the helmet transforms the wearer into a physical cyborg as the of man-machine is literally brought to life when the performers head is replaced by a computer monitor. Mashiro Mori's "uncanny valley" argues that the human reception of robots and other non-human forms is non-linear, Figure 20. As a robots shape approaches human form there exists a dip in which the differences between real and unreal forms are found disturbing and even repulsive [16]. In masking the human face, the robotcowboy helmet wipes away part of the wearer's humanity and fills it with computer output — a thousand and one faces. As a "humanoid form" the performer falls into the upper slope of the uncanny valley becoming a technological puppet, a walking computer. Since the helmet is constructed to hide the face as much as possible in order to foster this transformation, viewers can easily engage in the "suspension of belief" in which the actions and movement of the helmet's wearer become unreal.

Experimental performances undertaken utilizing the robotcowoy helmet:

- **debut**: an experimental cyborg performance using guitar, electronics, and sequencing software, April 2006 (Figure 21)
- **recharge**: an homage to Paik's *TV Buddha*; installation/performance, May 2006 (Figure 22)
- **midi_karaoke**: a living karaoke machine, November 2006 (Figure 23)

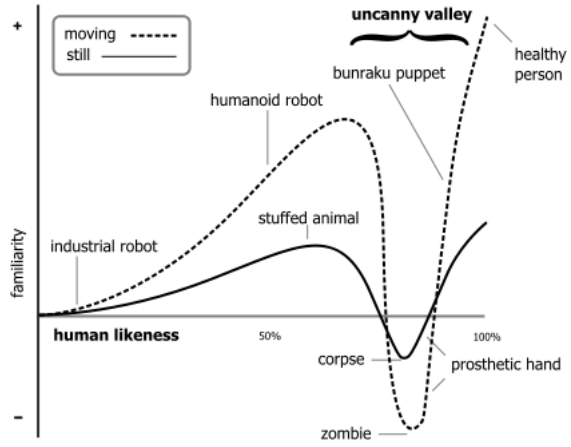


Fig. 20: Mori's ucanny valley

Fig. 21: *debut*, the first performance with the robotcowboy helmet, 2006

Fig. 22: *recharge*, an homage to Paik's *Video Buddha*, 2006

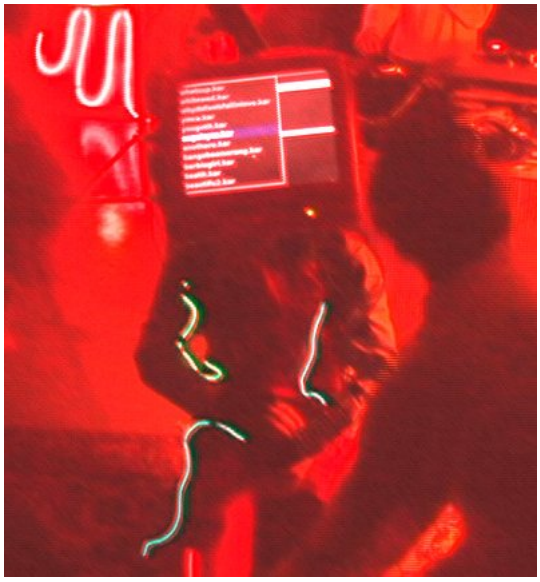


Fig. 23: *midi_karaoke*, a living karaoke machine, 2006

As a symbolic device, the helmet represents both the benefits and liabilities of augmenting oneself with technology. As with any new and interesting consumer devices, the helmet has always generated a certain amount of excitement, attention, and spectacle — at a dance party guerrilla performance, for instance, random women left over 10 lipstick imprints on the screen. Battery life and weight become important liabilities for the performer since, for example, if the video goggles run out of power, the wearer is blinded. Electrical energy concerns thus become integral to the helmeted performer as with any augmented cyborg.

The monitor helmet has become an icon of the robotcowboy project and the author, although at the time of writing it has not been used in over 6 months due to problems with the prototype. Obviously, weight is an issue and since the lcd is an older, cheaper model, it has a heavy, power hungry back light. The video goggles and 375 line video camera are far too low quality, making vision nearly impossible in low-light conditions and, during one performance, the automatic gain circuitry of the camera oscillated the brightness of the picture due to a strobe light. These issues can be rectified with a newer, lighter screen and better sighting system such as a periscope, fiber-optic cable sight, or more expensive video system.

As an experiment into human-computer performance, the prototype helmet provides a novel experience for the audience at a loss to the performer. It creates a believable cyborg avatar and is a good example of device art, yet inhibits the performer's mobility, vision, and performativity both symbolically and physically. As a costume element for a musician, the helmet is restrictive in that the instruments cannot be seen which requires one to practice blindfolded in order to be able to play effectively. Although begun in the auspices of a human-computer musical project, it was quickly realized that, although the helmet is effective visually, it is not as useful musically.

4.2 robotcowboy button_box

The robotcowboy button_box, Figure 24, is a wireless button in the triangular shape of the common green "play" symbol which gives a performer symbolic control of abstract musical computation. The box consists of button and light emitting diode (led) handled by a micro controller which utilizes an RS-232 serial Bluetooth modem for wireless connectivity. "cue" (play/pause) and "load" events are determined by the length of the button press: a short

Fig. 24: The robotcowboy button_box, 2006

press is a cue and a long press (greater than 1 sec) is a load and the led flashes during the loading process. Custom application and play list scripting receive these events and execute the corresponding action within the music software, in this case FL Studio. FL Studio[11] is a commercial sequencing and sound generation application which does not contain extensive automation controls, requiring non-ideal direct intervention using a keyboard and mouse during live performance. The author has performed solo guitar and vocals with computer backing tracks composed and played in FL Studio and the button_box is the result of these experiences.

A solo musician accompanied by electronic means is encumbered by forced interaction within a live setting. In the authors experience, onlookers can sometimes perceive that the electronic sound generation/playback device ways is in control of the performer — in between songs the solo guitarist must turn away from the audience and use his computer to set up the next song. A performance can be seen as a feedback loop of energy between the audience and musician and the experience can be diminished if this flow is interrupted over and over again. By using a wireless control device and automaton, the button_box becomes the only physical technological device on

[11] http://www.fruityloops.com/

stage as the computer can be hidden from view to break the focus on it as a needy performer. The green "play" button becomes a recognizable symbol of the wearers authority over technology and, mounted on the chest with velcro, this symbol is literally integrated into the body — the main focus of live musical expression.

The system worked successfully and was used in a number of performances. The Bluetooth modem provided a range of over 30 meters at the cost of a 30 minute battery life. Visual feedback from the status led and audio tones enabled the author and the audience to relate the action of pressing the button to sonic results: several tones were played and the led flashed as the song was loading. This small feedback is vitally important to the performer as it denotes the working status of the system.

In the scope of the development of this thesis, the button_box helped define several of the musical goals of the project. Although the box solved the important performance problem of the laptop onstage, it did not change the method of making music. Sequencing[12] is a very powerful technique utilized by numerous commercial hardware and software systems. For the author, however, the act of creation through sequencing is a laborious task which does not allow much room for spontaneity in the end result, as the score is rigidly followed each playback. Even though symbolically the music is being generated in realtime from a prearranged score within the computer, there is no aesthetic difference between this action and a prerecorded version of the same piece. As per Croft's definition[13], the liveness is merely procedural, not aesthetic. In order to escape this method of creation and performance, the author decided to seek his own software paradigm for sound scoring and generation within the robotcowboy project.

[12] creating a musical score through event timing and repetition of musical patterns
[13] See Section 2.1

5 robotcowboy unit

The robotcowboy unit is the wearable computer system at the core of the robotcowboy project. It is the machine half of the musical cyborg which handles realtime audio generation, controller mappings, and musical scores in the form of software patches. The main design requirements and goals of the system are listed and its description is broken down into the hardware and software implementations. *Velocipede*, a prototype performance mapping/song patch is presented in order to detail an example software score implementation and hardware interface. Results analyzing how well the system meets the design requirements and goals are discussed.

5.1 Design Requirements

Within the design space of this thesis project, it is important to define a limited set of requirements in order to narrow down particular points of focus. These were chosen through experimentation [14], from critical research [15], and for practical and aesthetic reasons. They are centered on the needs of the wearer in the live environment since the author desired a system that provides the limitless sonic potential and control of digital computation, yet is a easy and effective for performance and spontaneous creativity — just as analog instruments such as the electric guitar. This is the manifesto of sorts for the robotcowboy project but it can and should be applied as a basis for other electronic instrumental projects.

mobility the system should be as mobile as possible, allowing the performer to interact with the audience physically; the world outside of the concert hall becomes a venue through batteries

performance the system should not impede performativity, but enable it; previous computing systems have largely inhibited performance through forced interaction and special care must be maintained in order to avoid this pitfall; all interfaces and devices used by the system must be designed with live performance in mind; the musical aesthetic must not be sacrificed for the sake of "cool" technology, virtuosity and playability should *not* be subject to interaction

[14] see Section 4

[15] see Section 2.1

instrumentality in order to achieve instrumentality and performance excitement, the system must be able to produce "bad music" and have some capacity for failure, it should not "play itself"; a proper balance must exist between complexity and musical depth, it should be easy to pick up but hard to put down; tangible physical effort should be required in order to facilitate a more intense relationship between instrument and player; physical action and musical response must be mapped in meaningful ways with little delay to ensure embodiment of action and sound[16]

improvisation the system should enable improvisation and the "jam" session without forcing the simple repetition of playback; flexibility for different modes of song or theme performance should be maintained

reliability the system should just "work" and all elements must tested thoroughly before extended use; a live performance environment can be demanding and the system must be hardy enough to withstand it

low cost the system should be as low cost as possible without significantly sacrificing any of the other goals; most musicians do not have unlimited funds to spend on expensive technology, and, in the spirit of post-digital[17], do-it-yourself, and hacker culture, devices should be based off of cheaper, available items; why reinvent the wheel, when you can adapt a pre-made one?

5.2 Hardware Implementation

The hardware implementation of one-man-band cyborg apparatus that is the robotcowboy unit consists of a wearable computer, external soundcard, and peripheral devices. The computer is a specially designed machine for industrial use, the soundcard is a common digital audio interface, and the peripherals are built upon largely low-cost gamepads and joysticks.

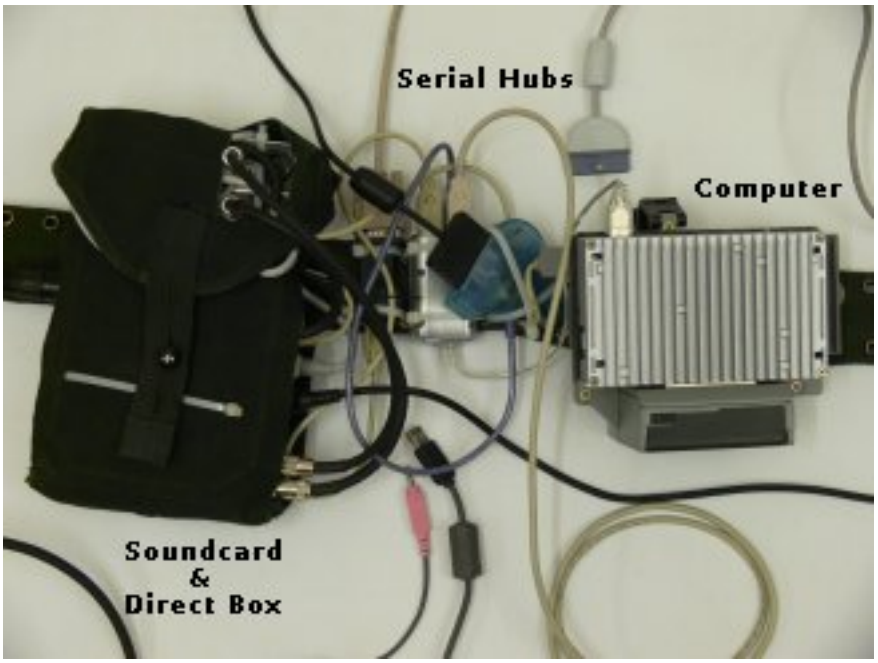


Fig. 25: The robotcowboy unit main hardware

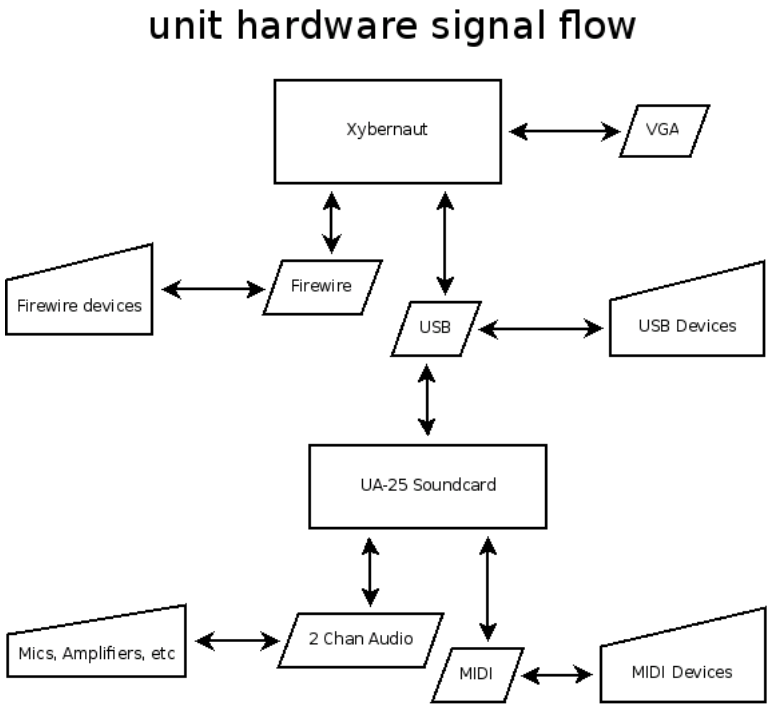


Fig. 26: The robotcowboy unit hardware flowchart

5.2.1 Computer

units main hardware (Figures 25 and 26) consists of a Xybernaut MA V wearable computer and a Roland UA-25 external Universal Serial Bus (USB) soundcard. The computer is a specially designed mobile computational platform for the commercial and industrial sectors with a low-voltage 500MHz Pentium 3 Celeron processor, 256 MB of RAM, a 4 GB hard disk drive (which has been upgraded to a 20 GB model), compact flash card slot, powered IEEE 1394 Firewire and USB ports, and an LVDS digital monitor connector. An expansion bay provides further powered Firewire and USB ports, a VGA monitor connection, and belt mounting brackets. Two battery packs give 3 hours of operation and the whole system weighs only about 1.5 kg. Although out of production as the industrial market for mobile devices has shifted from wearables to wireless tablets, this particular computer was purchased through the eBay online auction system for $350 USD, substantially cheaper than building a custom made system.

The wearable platform was chosen for its mobility design and reliability. A system already designed to be worn on the body (see Figure 27) and used during active use by the wearer makes an obvious choice and the Xybernaut is a hardened, power efficient device that is meant to work in a rough industrial environment. Being over 4 years old, computing capability is comparable to modern PDA's. These devices, however, do not provide as robust a platform since they are designed for commercial use, do not feature graphics ports, and contain built in screens which further drain battery life. Since the separate touch screen of the Xybernaut can be removed, further power is saved and a main symbol and focal point of the human computing experience is thus eliminated.

Lumsden and Brewtser [30] define important interaction design requirements for mobile and wearable devices which are fundamentally different from those of desktop computers. So far, most mobile interfaces have been designed with the desktop GUI in mind even though the mouse, keyboard, and screen paradigm is not practical, a screen can take too much focus, size, and battery power for example. The authors define 3 principles for mobile interaction: safe mobility, the interaction must not impede the users navigation of the environment; physical awareness, the interaction must be appropriate for the physical situation; and task awareness, the best interaction must be

[16]see Jordá and Croft in Section 2.1
[17]See Cascone in Section 2.1

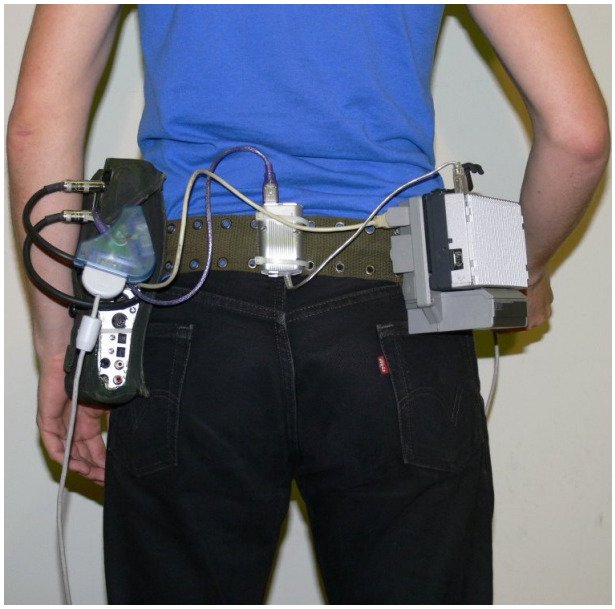


Fig. 27: The robotcowboy unit as worn on the body

chosen for the computing task at hand. For many devices, an emphasis must be moved away from the visual paradigm and toward the "hands-free" and "eyes-free" interaction for more effective specific use. By removing the display from the computer, the robotcowboy unit thus becomes an embedded mobile computer system whose interface can be more specifically designed to the needs of the musician. Its interaction is not mediated through a screen, keyboard, and mouse, thereby distancing it from the natural symbolism of the general purpose computing device:

> At a performance by Jim O'Rourke in 1997:
> *...there was no stage, Jim was set up at an ordinary table with his G3...halfway through his set, a young woman made her way through the audience to Jim's table, and began talking to him as he performed:*
>
> *Girl: "Hey, when's the next band on?"*
> *Jim: [distracted]"...uhhh, they're on right now."*
> *Girl: [confused]"Really? Who are they?"*
> *Jim: "...it's me!"*
> *Girl: [looks confused; walks away]*
>
> *...it was clear that the uncomprehending girl was thinking "that guy with the computer must be one of the organizers of the event"*

or something, that is, computers are tools for organizing information (and thus exercising control), but not for making art. [15]

Physically mounting the computational system on the body (see Figure 28) brings about a semiotic return to the embodiment of both electronic performance and sound. There is an obvious physical disconnect between a laptop musician and a machine that weighs him down through stationary design and forced graphical interaction. The laptop is a digital construct that requires users to adapt to it while the wearable computer can be seen as a post-digital device physically adapted to fit is wearer. It is, therefore natural that music created on laptops has been largely of the disembodied and "sonic-architecture" variants [15] since the performer cannot truly engage in an instrumental relationship with his computer[18]. Even when using separate gestural controllers, it is important to note the performer is subconsciously aware of this disconnect. In placing the computer directly on the body, this thesis projects hopes suggest a paradigm in which foster a physical, semiotic, and instrumental return to the body in the realm of electronic and computer music.

5.2.2 Soundcard

An external soundcard and a direct input box (see Figures 25 and 26) enable audio "plug and play" on stage. The Roland UA-25[19] USB bus-powered stereo soundcard features a microphone preamp, standard and optical audio connections, and MIDI input and output ports. An attached direct box converts high-impedance signals to microphone level for connection to a stage mixing and amplification systems [27] and both devices are mounted within a belt worn fabric case. This direct audio connectivity enables the performer to simply walk on stage and plug into an amplifier system, much like a guitar player plugging in his instrument. Mobility comes from this simple freedom by removing much of the required hardware setup (see Section 5.3.3 for a similar freedom from software setup).

Fig. 28: The author demonstrating the physical embodiment of the system

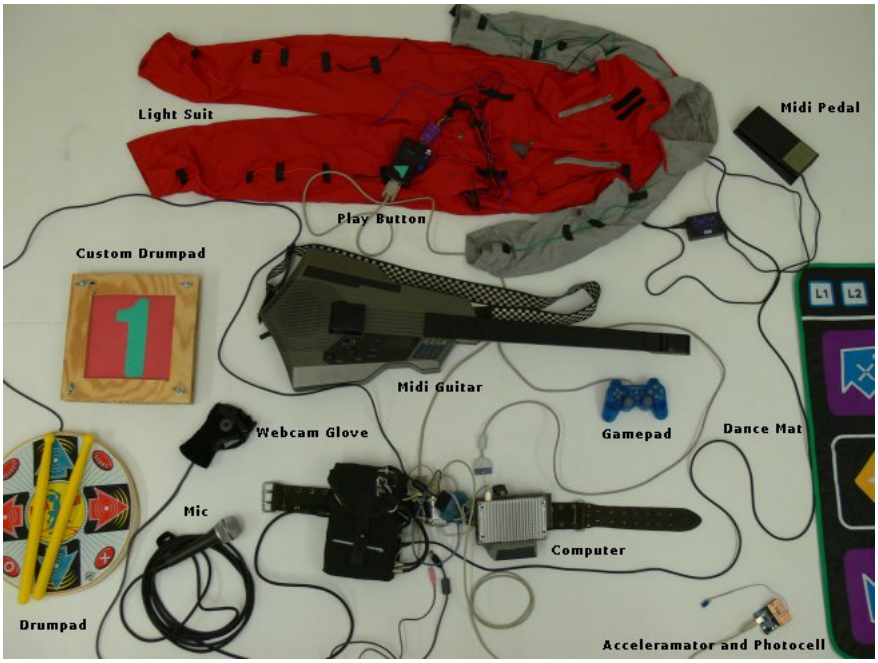


Fig. 29: Example robotcowboy unit peripheral devices

5.2.3 Input Devices

robotcowboy peripherals include human interface devices (HIDs), MIDI controllers, and serial devices (see Figure 29). unit is a platform for mobile computation and has no built in sensors or input devices of it's own and the built-in USB and Firewire ports offer the needed connectivity (see Figure 26) with low cost gamepads acting as stable sensor boards. USB to RS-232/parallel port converters can be used to connect older serial devices to the computer such as cheap micro controller systems and MIDI input/output is provided via the USB soundcard (see 5.2.2). Custom serial devices based on micro controllers can be incorporated such as the robotcowboy button_box (see Section 24) which is used to send play list and transport control commands. Currently, the main robotcowboy peripheral performance units thus far are common human input devices: a gamepad and a MIDI guitar (see 5.4 for more details).

As discussed by Steiner, Merill, and Matthes, [2] HID's provide easy access to gestural data through motion, pressure, and button presses and many

[18]see Jordá and Croft in Section 2.1
[19]http://www.roland.com/

Fig. 30: *PureJoy*: a live sampling and looping system [1]

have a high enough resolution and sampling rate for live musical performance. Since these devices are commonplace, their physical performance, gestural, and musical mappings are more transparent to an audience used to interacting with such interfaces. An increasing "physical computing" interest is looking to HID's for electronic sound, music, and video controllers as opposed to the creation of custom hardware which require a greater degree of technical expertise.

USB gamepads were chosen as the main interface platform since they are available, cheap, and easy to modify. As noted by Jensenius, Koehly, and Wanderly [22], such devices can be found almost anywhere for little over $20 USD and are far less costly and complicated then a comparable micro controller or specialized sensor interface. In many cases, the off-the-shelf interface is suitable for the performance requirements as in the example of David Merrill's *PureJoy* system [1] which controls a Pure Data live sampling and looping patch (see Figure 30). In others, a more customized device can be constructed from a gamepad by opening and wiring external switches and simple resistive sensors to the 10-12 digital buttons and 4 built-in 8 bit analog to digital converters (ADCs). Most analog sensors, such as Piezo discs, flex resistors, and motors, can provide interesting input options and some more complex system-on-a-chip devices, such as the QT113 capacitive touch sensor[20], send simple binary triggers which can be mapped to button events. Jensenius et al. demonstrated that even cheap conductive materials such as

[20]http://www.qprox.com/downloads/datasheets/qt113_105.pdf

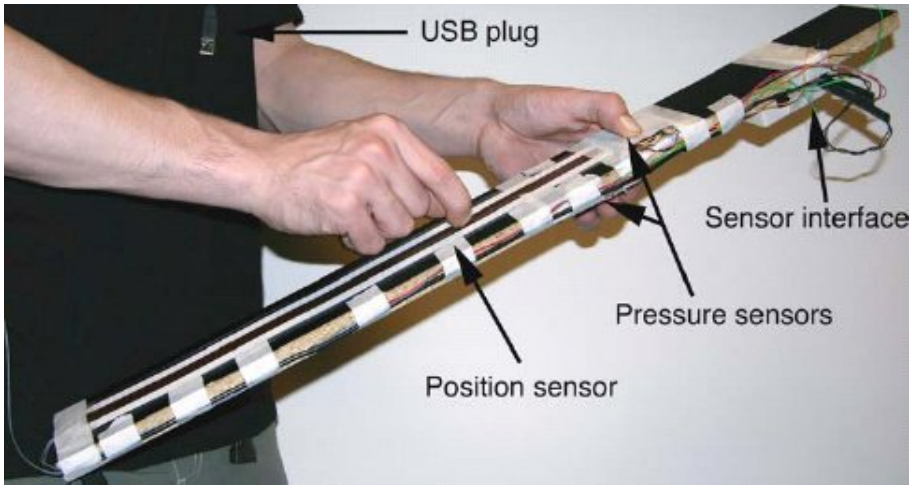


Fig. 31: *Cheapstick*: an affordable positional controller [22]

foam and magnetic video tape can be used to construct a custom positional controller: the *Cheapstick* [22] (See Figure 31). Specialized software drivers are required to map joystick and gamepad events to musical parameters and this is accomplished within the robotcowboy project using the unit-daemon (see Section 5.3.3). Dedicated modern video game console controllers such as those for the Playstations 1-3, Gamecube, and Xbox/Xbox 360 are very durable and can be used on regular computers through third-party adapters. Examples of planned robotcowboy interfaces listed below represent a small amount of the possibilities for custom interaction:

- A 16 button matrix made by wiring mechanical push buttons to the gamepad's original button contacts
- 4 potentiometers mounted on different areas of the body wired to the gamepad's 4 ADC's
- Resistive bend sensors for one hand wired to the 4 ADC's
- A spinning weight attached to a motor whose speed is read by an ADC
- A glove with simple capacitive touch sensors, QT113's, connected to optoisolators which trigger buttons
- Piezo discs worn on the feet which are read by the ADC's for 'stomp' sensing

By choosing a low-cost and readily-available input platform, a greater number of varied instruments can be constructed and used in combination. It is the essence of the post-digital instrument to reuse and adapt technology, as stated by Cascone and Richards in Section 2.1, and this aesthetic choice can lead to greater creativity than the use of much more advanced yet bulky and almost prohibitively expensive interfaces such as the $650 IRCAM Eobody[21]. By doing some simple math, $650 / $20 = 32.5, it is obvious that having 32 possible instruments is preferable to one interface box without additional sensors and this sum, in fact, approaches the entire cost[22] of the robotcowboy unit! Granted, the 100Hz sampling rate and 8 bit resolution of most gamepads are not as flexible as compared to the 200 - 4000 Hz and 7 - 16 bits of many popular sensor interfaces, but the author for one does not mind this trade-off. [22]

[21] http://www.forum.ircam.fr/361.html?&L=1

[22] $720, see Section 6 for more details.

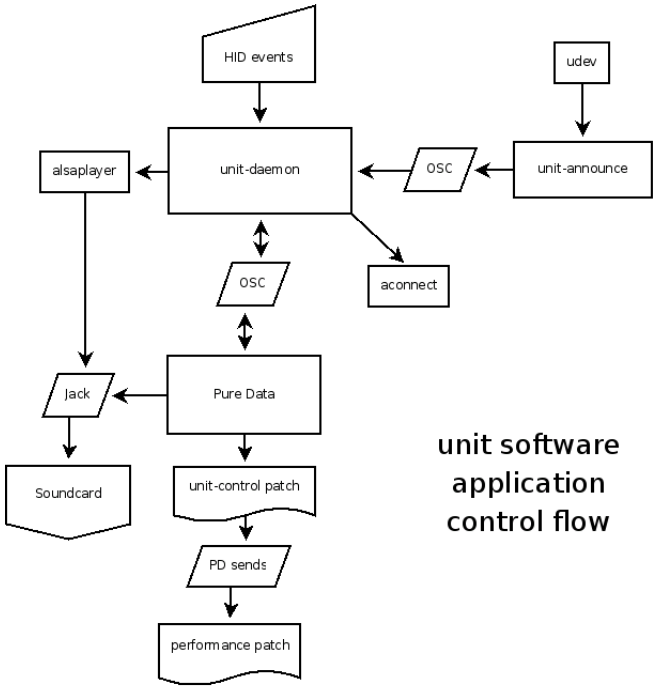


Fig. 32: The robotcowboy unit software flowchart

5.3 Software Implementation

The software which powers the robotcowboy unit wearable computer is a custom input daemon, the Jack realtime audio daemon, and Pure Data running on GNU/Linux (Figure 32).

5.3.1 Operating System

GNU/Linux was chosen as it is a free, stable, and easily customizable operating system that can be slimmed down to run efficiently on slower hardware. The Linux kernel[23] running the robotcowboy unit, for instance, has been specially compiled for realtime audio applications. GNU/Linux has a proven track record in embedded devices, servers, and a growing desktop user base. The philosophy of the Free Software Movement [24], whose members developed GNU/Linux, is that software should be free and its source code openly dis-

[23]the main operating program of the GNU/Linux operating system
[24]http://www.fsf.org

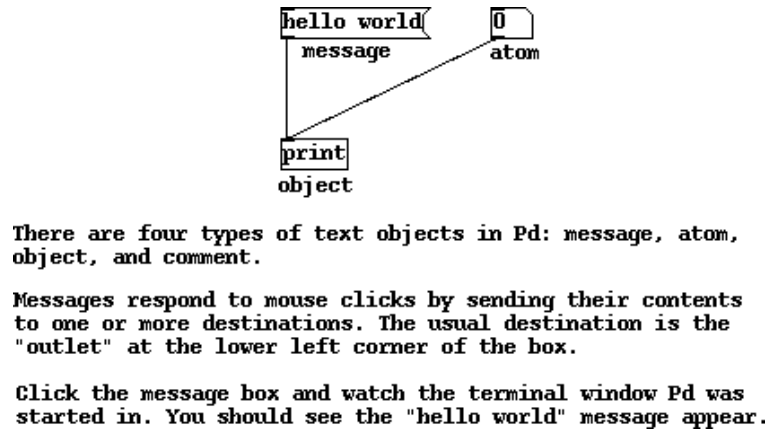


Fig. 33: An example Pure Data patch

tributed which encourages collaboration and creativity among its users. The software environment is designed for programming and application expansion and this sharing of information and customization allows users to construct free, stable systems for specific needs.

5.3.2 Sound Generation and Processing Environment

Pure Data (or Pd) [37] is a graphical "patching" program and object-oriented interpreter for realtime audio processing created by Miller Puckette in 1996 as a logical progression of MAX [36]. It consists of a realtime software digital signal processor (dsp), event scheduler, and a separate gui (graphical user interface) that presents atomic elements and user made modules as small boxes, or objects, which are connected, or "patched", together showing a representation of the signal processing flow (see Figure 33). As Martin Dixon notes [13], it is this graphical abstraction and throw back to modular analog patching synthesizers and processes that enables creativity, "play", and improvisation with largely complex digital signal processing methods. The very interaction design of Pure Data encourages the post-digital desire to tinker and create empirically.[25]

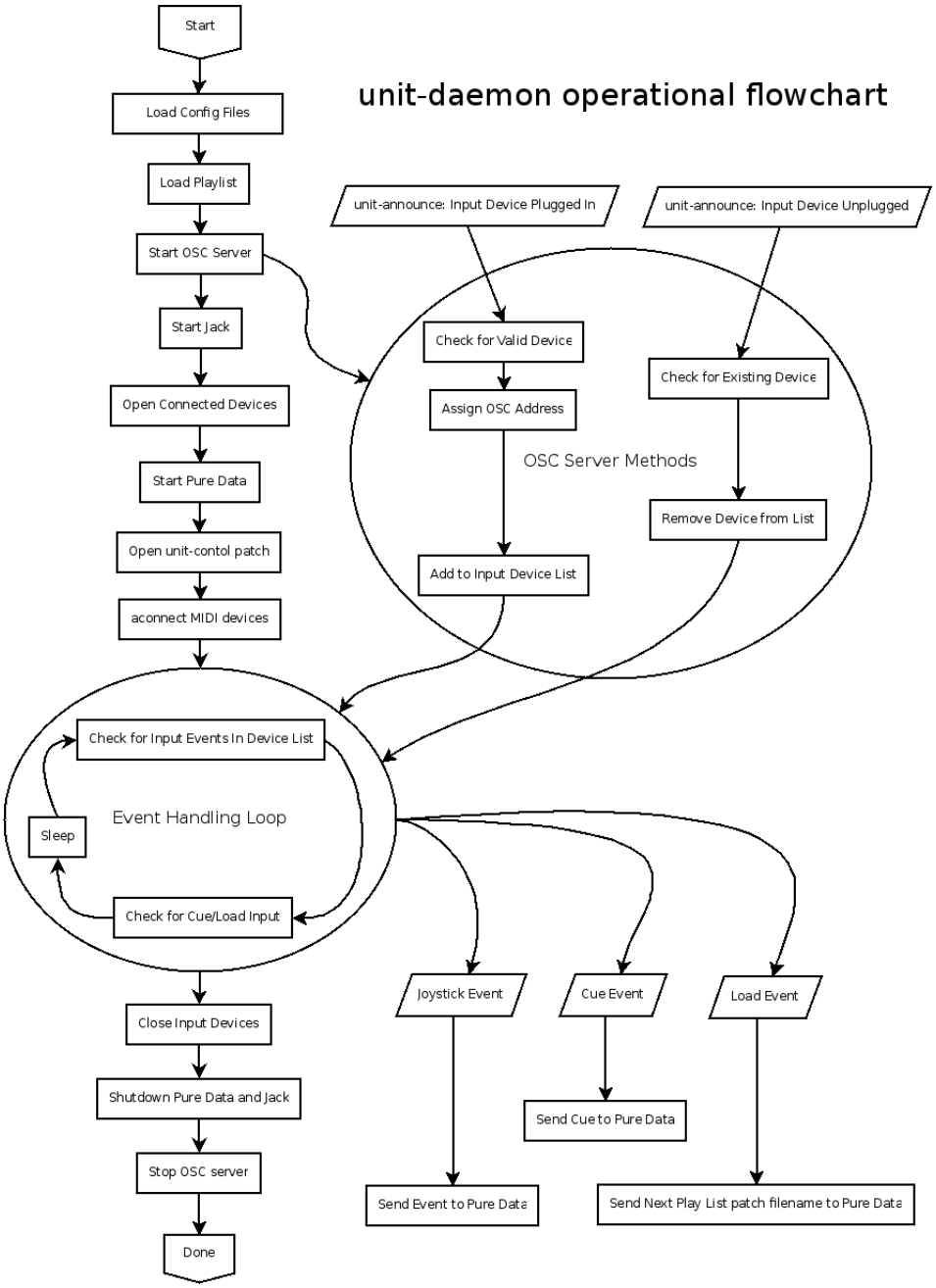


Fig. 34: unit-daemon flowchart, see unit-daemon.cpp in Appendix A

5.3.3 unit-daemon

"unit-daemon" is a custom input and play list handling daemon written in the C++ object-oriented programming language which essentially scripts all of the separate software elements of the system[26] (See Figure 34). It automatically starts the Jack realtime audio daemon[27] , initializes Pure Data and sets it to use Jack, calls the Advanced Linux Sound Architecture [28] program "aconnect" to route MIDI between Pure Data and the external soundcard, and enters an input device handling loop. Open Sound Control (OSC)[29] is used for device communication between unit-daemon and a Pure Data control patch, unit-control (Figure 35), which accepts OSC commands to open, close, start, and stop specified patches which are kept in a circular play list for live performance. All control aspects are customizable in configuration files[30] such as the commands that start Pure Data and Jack, the soundcard to be used, the control patch to load, and the play list patch filenames. Specific event feedback is provided to the user through audio playback using the alsaplayer application. Although specifically designed to use Pure Data, unit-daemon can be easily extended to provide control over other realtime sound software such as MAX/MSP[31], Supercollider[32], and ChucK[33].

unit-daemon automatically handles the insertion, use, and removal of USB joystick devices. Much like the Pure Data HID-toolkit [2], it consists of an event listener within the unit-daemon main loop and a device hot plug mechanism to notify the listener. Custom rules created for udev[34], the device manager for modern versions of the Linux kernel, call unit-announce[35], a small OSC application which notifies unit-daemon, when a joystick device (/dev/input/joy*) is connected to the computer. OSC server methods within the daemon automatically mount the device and check its USB device name against a configuration file[36] which can specify an OSC sending

[25]See Section 5.4 for a prototype robotcowboy performance patch
[26]The source code is available in Appendix A
[27]http://jackaudio.org/
[28]http://www.alsa-project.org/
[29]http://opensoundcontrol.org/
[30]See the configuration files in Appendix A
[31]http://www.cycling74.com/products/maxmsp
[32]http://supercollider.sourceforge.net/
[33]http://chuck.cs.princeton.edu/
[34]See Appendix A for the rule file
[35]Source code available in Appendix A
[36]See joystick name configuration file in Appendix A

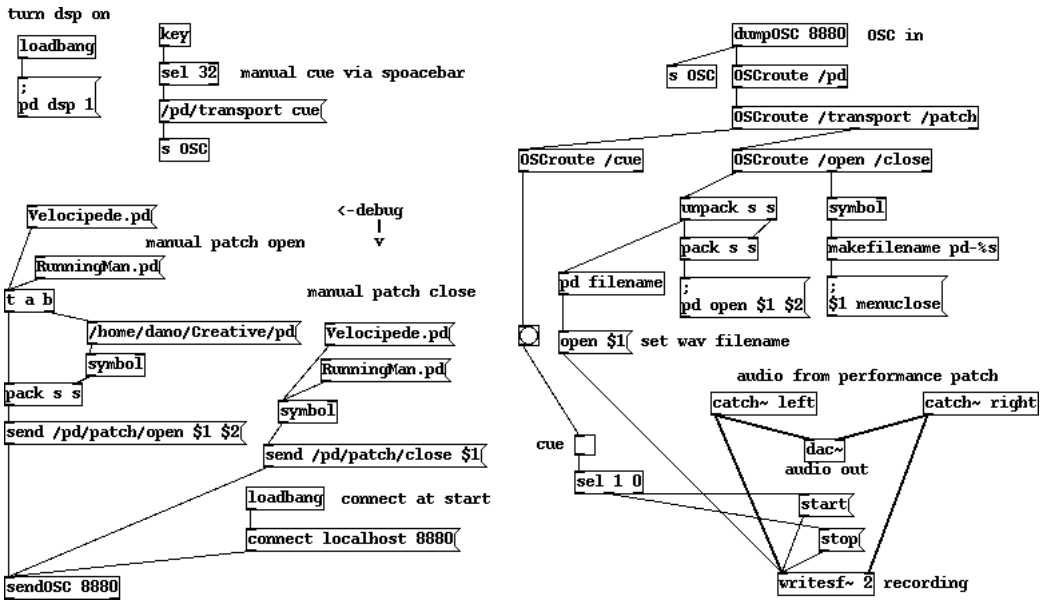


Fig. 35: unit-control: opens/closes performance patches and acts as a routing gateway for instrument and control events

address to associate with the joystick or gamepad. Once registered, each device's button and axis events are read within the event handling loop and sent to the Pure Data control patch via OSC which then routes the events to child performance patches. When a joystick or gamepad is removed, unit-daemon is notified once again by udev and cleanly closes the device. Thus multiple controller "plug and play" is handled by the the daemon software while the specific input mappings and device addresses are defined within each performance patch.

unit-daemon essentially solves the logistics problem of controlling multiple software applications on a mobile device, allowing the performer to focus on the performance and not the mechanics of cuing tracks and setting up input devices. It eliminates the need for a graphical gui and requires only one button for track and transport control which can be nearly anything: a hacked keyboard controller, gamepad, specialized serial box, etc. By automating device setup and mapping within performance patches, the daemon transforms complex digital devices into reliable "plug and play" instruments that work when plugged in, much in the same manner as an electric guitar when connected to an amplifier. By simply moving logistical software needs away from the performance and into the setup phase, the musician can now view his device as a responsive instrument, not a digital construct that

requires constant attention and care.

5.3.4 Interaction and Recording Affordances

Since the robotcowboy unit computer lacks a screen (see Section 5.2.1), audio feedback is used to provide "eyes-free" operation [30]. The robotcowboy software interaction is designed to confirm physical actions taken by a user such as the insertion and removal of input devices and the loading of patches within the play list. Since sound is the main output of the entire system, sound files are played upon these events as well as when a problem occurs so as to notify user of the current software actions and states. Devices, such as the robotcowboy button_box (see Section 4.2) can add a simple visual output in the form of an led and the small motor-driven counter weights of "rumble pad" gamepads can provide a tactile element. This feedback is essential as the performer must know when the system is functioning correctly.

Recording musical performances can be a tiring and expensive task for most musicians and the robotcowboy unit software remedies this problem by directly recording stereo mix and/or separate track .wav files within Pure Data whenever a performance takes place. A 1 GB compact flash card provides ample room and ease of transfer to a separate computer with a card reader. The live performance can then be distributed directly to the audience via compact disk or electronic means thus marking the recording's musical and temporal location by linking it to a definable tangible experience. If every performance is recorded, a catalogue of improvisations and works can be built up for later perusal and selection thus removing the pressing need to record "for posterity". Once again, a small freedom given by the software can allow the performer to focus on the performance at hand since the "perfect" rendition will never be missed.

5.4 Velocipede: A Prototype Performance Mapping

Velocipede is this thesis' most developed prototype interface mapping/musical score and is described in order to offer an example of a robotcowboy performance instrument. The physical input device is a Playstation 2 Dualshock game controller (Figure 37) connected through a USB adaptor and assigned an OSC sending address at "/pd/devices/ps2black". A Pure Data patch de-

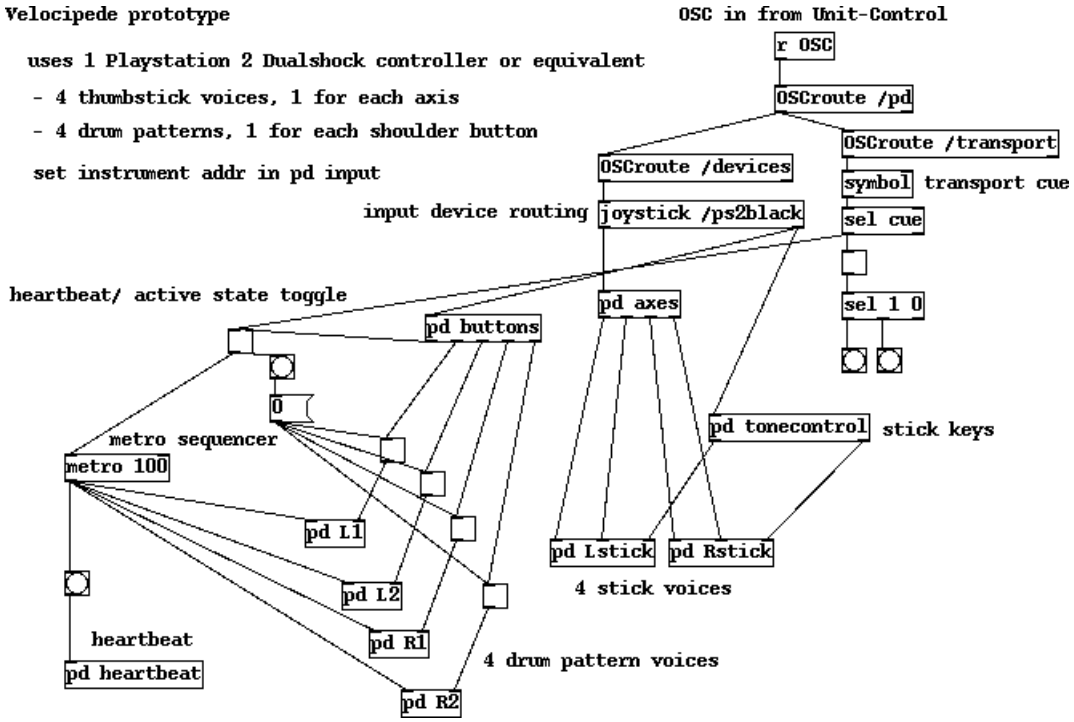


Fig. 36: The *Velocipede* Pure Data patch

Fig. 37: The *Velocipede* Dualshock controller

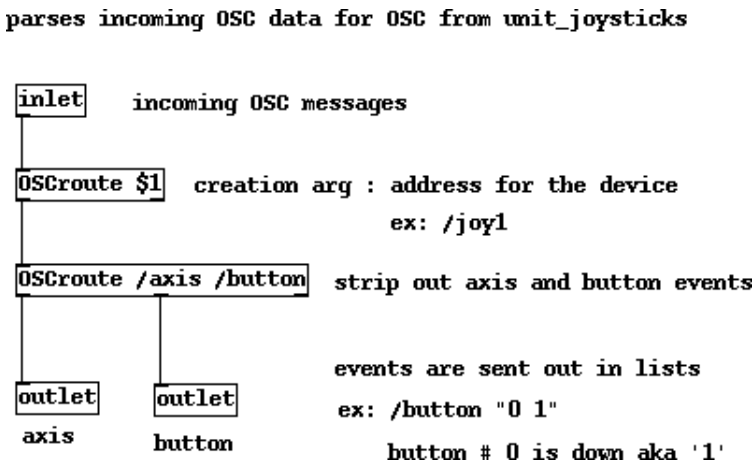


Fig. 38: joystick: routes the button and axis events from a specified input device OSC address

fines the software "score" — the instrumental and musical mappings of the controller. The main patch (Figure 36) consists of:

- OSC transport control for cues sent to the "/pd/transport" address
- input and event routing subpatches
- tone control and sound generation for the Dualshock's two analog thumbsticks
- four rhythmic sequences triggered by the shoulder buttons
- a heartbeat metronome toggled via the start button or transport cue

The joystick object, ie. the box labeled "joystick /ps2/black", (Figure 38) routes the button and axis events from the Dualshock and the axes and button subpatches (Figures 39 and 40) route the required axis and button events respectively to the mapping objects.

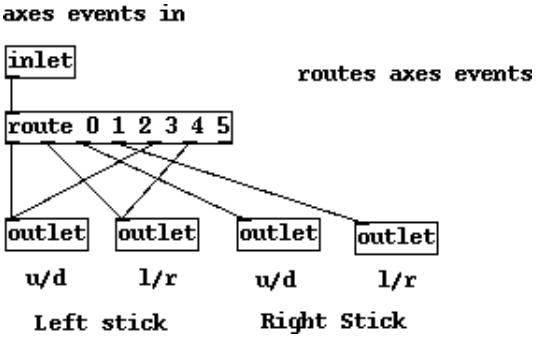


Fig. 39: axes: routes axes events

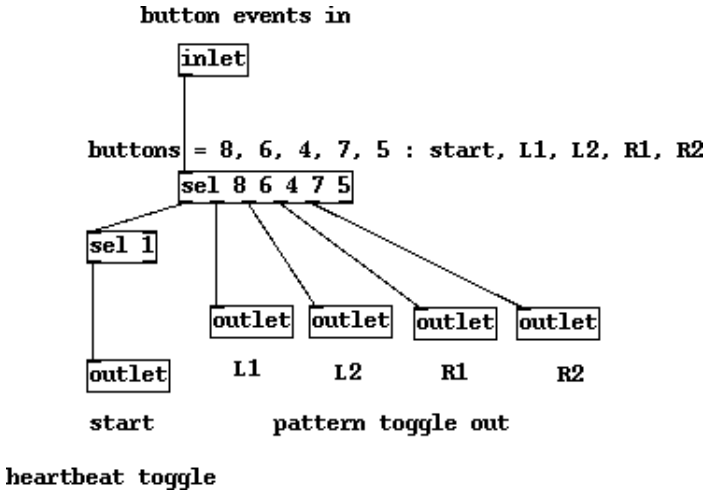


Fig. 40: buttons: routes button events

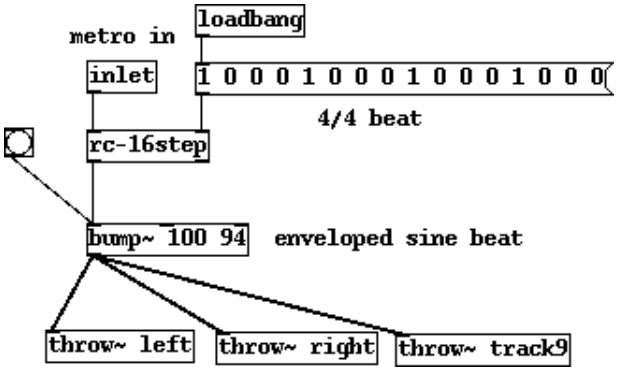


Fig. 41: heartbeat: plays a 4/4 heartbeat when the patch is active

The main musical content is defined within the stick, tonecontrol, and shoulder button subpatches and heartbeat (Figure 41) provides a 4/4 beat metronome when the patch is cued and running. These mappings are represented in Figures 42 and 46. Lstick (Figure 43) and Rstick map the x and y axis of each stick to a "midi_sawosc" additive synthesis sound object on a single channel and the incoming axis values are scaled from -32767 to 32767 down to 0 to 30. Once within the MIDI range (0-127), these values are added to offsets (30, 40, 61, 71) provided by tonecontrol (Figure 44) which is toggled via the two buttons beneath the thumbsticks. As a result, the center value of the left stick defaults to MIDI note 45 (30/2 + 30) and is always 31 steps below the right stick whose value is 76 (30/2 + 61) — in essence, they can be thought of as bass and treble. When the right thumbstick button is depressed, both values are increased by 10 to 55 and 86 and the left thumbstick button returns them to 45 and 76, yielding two "keys" in which the thumbsticks can be played. The pitches of each axis are mapped in a right-hand Cartesian fashion with a maximum and minimum of +/-30 from the center value: the pitch increases in the top and right directions and decreases towards bottom and left. Each set of voices, two for each stick are then mapped to specific channels with the top/bottom axes panned completely to the left and left/right axes panned to the right.

Each of the four shoulder buttons triggers a drum sequence with its own subpatch. The L1 subpatch (Figure 45) triggers a simple 16 step sequence by opening a spigot to the heartbeat metronome when it receives a '1'. The 16 beat sequence "1 1 0 0 1 1 0 0 1 1 0 0 1 0 1 0" plays the "pish" sound object, a simple enveloped pink noise snare drum. 1's are triggers while 0's are treated as rests and the sequence is reset back to its initial position when the button is released. L1, L2, R1, and R2 have distinctive patterns and

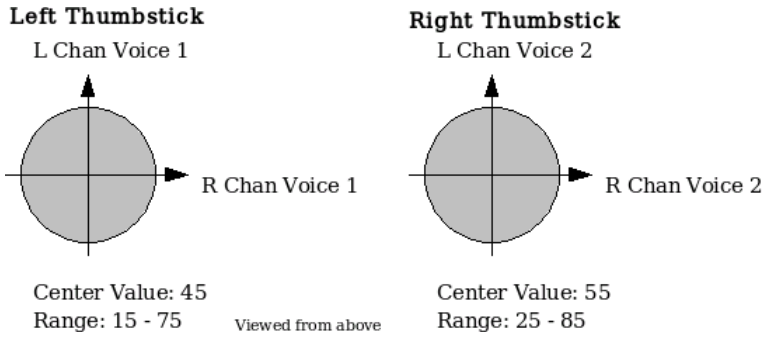


Fig. 42: The thumbstick pitch mappings with default center values

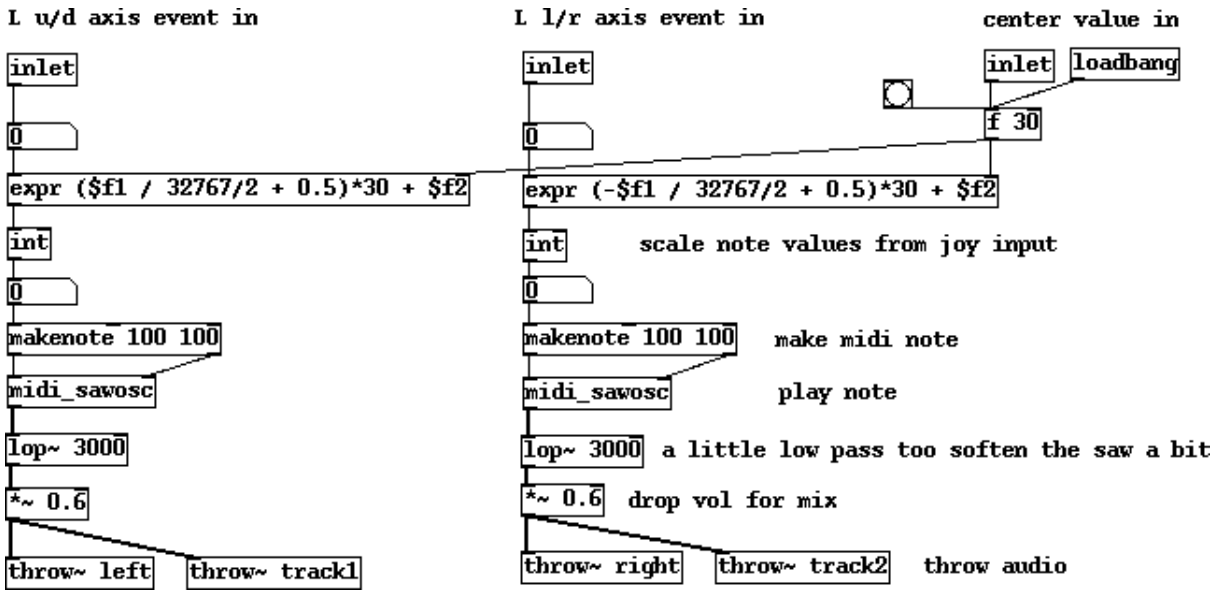


Fig. 43: Lstick: scales the left thumbstick axis x and y values into MIDI notes

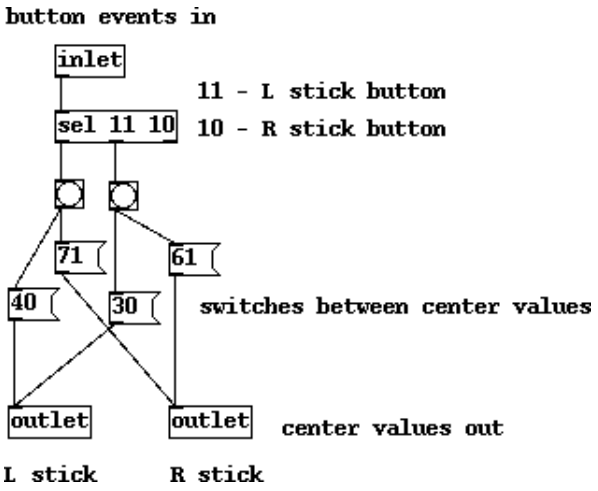


Fig. 44: tonecontrol: changes the center MIDI note for each thumbstick when either thumbstick button is pressed

timbral percussive sounds which can be triggered at will by the performer. Holding down a button will loop its pattern and, if the sequences are triggered asynchronously, interesting poly-rhythmic patterns emerge.

As a result of its design, *Velocipede* is a dynamic instrument that is both easy to use, but hard to master. The fine degree of control on the thumbsticks allows rather interesting microtonal expressive gestures and the panning of the axes gives each thumb control over two separate voices: by moving the stick diagonally to the lower right, the pitch in the left voice decreases while the right voice increases. Two sticks yield four voices in total and an interesting and meaningful relationship is drawn from the physical construction of the thumbsticks themselves in that they are spring loaded to return to their dead center position. If the sticks are moved and then "flicked" back to this equilibrium state, the separates voices in either channel suddenly snap back together in a return to stereo — both channels suddenly play the same tone in phase thereby combining and doubling the acoustical loudness. The effect is rather dramatic and when timed with the rhythmic progressions discussed below can be quite effective.

By contrast, the shoulder buttons offer a simple one-to-one mapping for the triggering of their sequences along with the main 4/4 heartbeat. By simply holding some of them down, poly-rhythmic relations develop naturally since each sequence is different and by utilizing the different timbral aspects of the enveloped white noise, pink noise, sine and square waves the

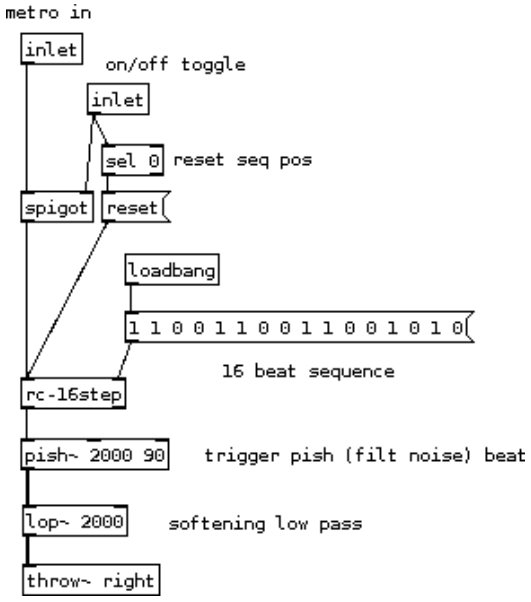


Fig. 45: L1: Triggers a simple rhythmic drum sequence when the L1 button is pressed

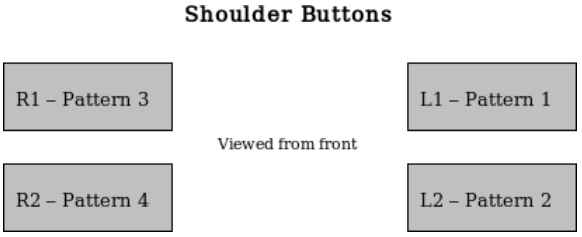


Fig. 46: The rhythmic control button mappings

performer can build an overall rhythmic gesture. Quick triggering can be used to play each sound individually, although this is harder to do while effectively handling the thumbsticks at the same time. Currently the tempo is rigidly defined within the patch and adding a tempo control mapping could expand rhythmic possibility.

All in all, this prototype demonstrates how the robotcowboy unit ties together the interface, mapping, and sound generation. Even a non-modified gamepad can yield a useful and engaging instrument when mapped to sound and musical gesture in a meaningful way. The ease of plugging in the specific controller and the described interaction quite naturally strongly associates the physical device with the patches sonic characteristics although this, of course, can be changed from patch to patch. After developing, playing, and performing with *Velocipede*, the author of this thesis can strongly assert that this performance device and musical mappings are indeed an engaging and exciting instrument to play.

6 Results

The robotcowboy unit system meets all of the design requirements set for it in Section 5.1, providing a platform for digital instrumentality built on the embodiment of action and sound. A computer worn on the body symbolically allows for physical movement and emphasizes an integration of man and machine in the spirit of the one-man band, a roaming musical cyborg with "the bass note in his left ear". By removing the forced interaction of desktop computing the system frees much of the musician's attention during performance so he can express himself and interact with the audience. The automation of logistical software tasks transforms the generic computing device into a specific embedded instrumental platform that "just works".

6.1 Mobility

Movement is restricted only by the audio cables (which can be eliminated through a stereo wireless audio link) and the performer can easily dance, gesticulate, and move out into the audience. A battery life of 3 to 4 hours provides ample running time for long outdoor sets using a portable speaker system. The only point of concern is the mechanical hard drive which can be damaged due to excessive shock and will be replaced by a sold state flash memory hard drive once the prices on these drives becomes feasible.

6.2 Performance

The stage aesthetics of the performances using unit become much more intimate and exciting than comparable laptop performances since the musician can move out into the audience directly. The physical action and musical response are much more transparent when the audience can see what the performer is doing. Devices can be developed for collaborative musical expression with the audience becoming part of the experience directly. There is no austere separation, no physical screen in the way, and the performer is now able to step down from the stage and get sweaty.

6.3 Instrumentality

The computer system itself is essentially a platform for the devices and software which create the sound — it is the medium, not the message. The message comes from the connection of physical interface and musical parameters which in turn is what makes an instrument. Since the physical interface is no longer the sound producing body the computer is required to transform the physical input to an amplifying system. If these mappings are well designed, the relationship can become so well connected that the device becomes the symbolic sound producer. By providing a generic basis for instrumentation and musical mapping, the physical devices and sound patches used in the robotcowboy project are separated offering modular reuse and an increase in overall performance and musical possibility.

6.4 Improvisation

Velocipede, as an example of a robotcowboy instrumental setup, does not script any of the musical parameters beyond several simple drum patterns. There is a great freedom of expression that can result in a vast number of different performances. A performer can easily "mess up" and produce "bad music" and this capability (or limitation, depending on viewpoint) for failure is one of the basic requirements for an exciting live performance. The inherent "jitter" of the human muscular and nervous system imparts an interesting offset during live performance and, by the design of *Velocipede*'s mappings, the lack of quantization allows for great musical possibility. The author has performed live with the system and can guarantee its capability for what many consider an engaging performance.

6.5 Reliability

As a result of extensive design, experimentation, and testing the entire system "just works". Flip the switch and, after booting up, a voice announces "unit ready". Plug in a performance device and a voice acknowledges that the device is registered by the system and sending events. Press start and away you go. The rugged hardware is designed for action and mobile use and custom devices will be encased and shrink wrapped to ensure reliability.

6.6 Low Cost

A cost break down demonstrates that the entire system was built for a relatively small amount:

Item	Price (in USD)
Xybernaut MA V wearable computer	$350
Roland UA-25 USB soundcard	$250
USB hub	$20
gamepads, USB adaptor, and other peripherals	$100
operating system and software	$0
Total:	$720

$720 for a viable and extensive human-computer performance system is far less expense then many of the projects listed in the previous works in Section 3.2. In fact, this sum is enough to purchase a cheap laptop computer and a majority of savings comes from the choice to use free, open source software. As the system is expanded with more and more devices its cost will increase through with the use of more expensive sensors and technology, but the initial price so far is definitely affordable. By aiming at low-cost, the system is affordable to those who pursue the post-digital aesthetic in its essence – hackers and experimental musicians in the non-academic environment.

7 Future Work

There is more work to be done with the robotcowboy system. It will be taken on the road where new performance devices and software mappings/patches will be developed. The unit-daemon logistical software will be released as open source code so as to allow others to experiment with musical device and software automation. As new devices emerge, the robotcowboy unit will be expanded to encompass their new musical and gestural possibilities. Last, network performances will be developed for use with multiple computer performers using the robotcowboy software.

Now that a stable system has been demonstrated, devices and performance patches will be built and taken on the road. The robotcowboy project does not aim to produce disposable prototypes, but instruments for long-term use and the best way to develop and test such devices is in the field — the stage and street. It is hoped that through demonstration, the embodied performance paradigm of the robotcowboy unit will be taken more seriously in the realm of live electronic music and digital instrumentation. In fact, new devices and mappings may arise through the insights gained from extensive use and experience.

In the spirit of free software, the logistical software daemon, unit-daemon, will be released as open source and made freely available. Lessens learned from this thesis' implementation will be incorporated into a rewrite of the essential core components in a cross platform environment such as Python so as to make the system useful to users of Microsoft Windows and Apple MacOS and the software will be packaged for easy installation. It is hoped that other performers may experiment with, expand, and adapt the software to enable their own embodied performance paradigms.

unit-daemon currently provides a platform for HID-based interfaces and can be expanded to include Bluetooth and wireless devices. The Nintendo Wiimote[37] is a good example of a Bluetooth gaming device which is being repurposed for numerous musical and gestural applications. As new standards of connectivity and interaction are developed, the system can be modified to handle these new devices in order provide greater gestural and performance possibility.

[37] http://wii.nintendo.com/controller.jsp

Since all of the inter-application communication within unit-daemon is handled within OSC, the software elements can be separated onto different machines over a network. A master/slave capability will be added to unit-daemon so multiple computers can send attached device events to a central audio processing station. Multiple wirelessly-networked wearable computers can be used to create an mobile electronic performance band.

8 Conclusions

This thesis set out to develop a live human-computer musical performance system, a technological one-man-band: the robotcowboy unit. unit offers musicians a wearable computational platform on which to develop physical controllers and software sound mappings, an adaptable electronic instrument built upon mobility and embodiment. The wearer of the system does not need anyone but his creativity and determination for an engaging performance. Through careful design decisions, the robotcowboy unit is an enabling computational platform as opposed to a laptop computer which subjugates much of a musicians ability to perform. It is not the aim of this thesis to target the laptop for ridicule, but to bring about a discussion to an alternative form of electronic instrument that can address the problems of instrumentality and mobility. It is hoped that the work of this thesis will inspire others to experiment with and develop such systems of their own — to join the ranks of the "robotcowboy".

References

[1] Jamioki-purejoy: A game engine and instrument for electronically-mediated musical improvisation. In *Proceedings of NIME-07*, New York, NY, USA, June 2007.

[2] A unified toolkit for accessing human interface devices in pure data and max/msp. In *Proceedings of NIME-07*, New York, NY, USA, June 2007.

[3] Laurie Anderson. *Stories from the Nerve Bible: A Retrospective, 1972-1992*. Harper Perennial, New York, 1994.

[4] L. Ashline, William. The pariahs of sound: On the post-duchampian aesthetics of electro-acoustic improv. *Contemporary Music Review*, 22(4), 2003.

[5] F. Bebey. *African Music: A People's Art*. Lawrence Hill, Westport, CT, 1975.

[6] Bert Bongers. An interview with sensorband. *Computer Music Journal*, 22(1), March 1998.

[7] Mark. A. Bromwich and Julie. A. Wilson. bodycoder: A sensor suit and vocal performance mechanism for real-time performance. In *Proceedings of ICMC 1998*, 1998.

[8] Mark. A. Bromwich and Julie. A. Wilson. Lifting bodies: Interactive dance - finding new methodologies in the motifs prompted by new technology - a critique and progress report with particular reference to the bodycoder system. 2001. Available from: `http://www.geocities.com/marekbuk/EDTlift.html` [cited June 2007].

[9] Kim Cascone. The aesthetics of failure: 'post-digital' tendencies in contemporary computer music. *Computer Music Journal*, 24(4), Winter 2000.

[10] John Croft. Theses on liveness. *Organized Sound*, 12(1):59–66, April 2007.

[11] G. Deleuze and F. Guatarii. *A Thousand Plateaus: Capitalism and Schizophrenia*, page 474. University of Minnesota Press, Minneapolis, MN, 1987.

[12] Maywa Denki. Tsukuba music. 2007. Available from: `http://www.maywadenki.com/concepts/what_tsukuba.html` [cited June 2007].

[13] Martin Dixon. Echo's body - play and representation in interactive music software. *Contemporary Music Review*, 25(1/2):17–25, Febuary/April 2006.

[14] Simon. Emmerson. Acoustic/electroacoustic: the relationship with instruments. *Journal of New Music Research*, 27(1-2):146–64, 1998.

[15] Joanne Favilla, Stuart & Cannon. Fetish: Bent leathers palpable, visceral instruments and grainger. *Contemporary Music Review*, 25(1/2), Febuary/April 2006.

[16] Terrence Fong, Illah Nourbakhsh, and Kerstin Dautenhahn. A survey of socially interactive robots. *Robots, Robotics and Autonomous Systems*, 42(3-4):143–166, March 2000.

[17] L. Gaye, R. Mazé, and L Holmquist. Sonic city: The urban environment as a musical interface. In *Proceedings of NIME-03*, Montreal, Canada, 2003.

[18] RoseLee Goldberg. *Performance Art: From Futurism to the Present*, pages 7–9. World of Art. Thames & Hudson, third edition, 2001.

[19] Chris Hables Gray and Steven Mentor. *The Cyborg Handbook*. Routledge, New York, 1995.

[20] Ian Holmes. *Electronic and Experimental Music*, page 23. Routledge, second edition, 2002.

[21] E. Huhtamo. *Cybernation to Interaction: A Contribution to an Archealogy of Interactivity*, pages 96–100. MIT Press, Cambridge, MA, 1999.

[22] A. R. Jensenius, R. Koehly, and M. M. Wanderly. Building low-cost music controllers. In *Proceedings of Computer Music Modeling and Retrieval 2005*, 2005.

[23] Silvija Jestrovic. The performer and the machine: Some aspects of laurie anderson's stage work. *Body, Space, Technology Journal*, 1(1), 2000. Available from: `http://people.brunel.ac.uk/bst/1no1/SILVIJAJESTROVIC.htm` [cited June 2007].

[24] Sergi Jordá. Afasia: The ultimate homeric one-man-multimedia-band. In *Proceedings of NIME-02*, Dublin, Ireland, May 2002.

[25] Sergi Jordá. Digital instruments and players: Part i efficiency and apprenticeship. In *Proceedings of NIME-04*, Hamamatsu, Japan, 2004.

[26] Sawako Kato. Digital media, and sound art. In *Proceedings of WFAE Symposium 2003*, Australia, 2003.

[27] Al Keltz. A direct box can be in-di-spensible. Available from: `http://www.whirlwindusa.com/ftp/tech/tech02.pdf` [cited June 2007].

[28] R. B. Knapp and H. S. Lusted. A bioelectric controller for computer music applications. *Computer Music Journal*, 14(1):42–47, 1990.

[29] Volker Krefeld. Biography of michel waisvisz. 2004. Available from: `http://www.crackle.org/short%20biography.htm` [cited May 2007].

[30] J. Lumsden and S. Brewster. A paradigm shift: Alternative interaction techniques for use with mobile and wearable devices. In *The 13th Annual IBM Centers for Advanced Studies Conference CASCON'*2003, Markham, Ontario, Canada, oct 2003.

[31] Akitsugu Maeybayashi. Home page. 2007. Available from: `http://www2.gol.com/users/m8` [cited June 2007].

[32] Akexei Monroe. Ice on the circuits/coldness as crisis: The resubordination of laptop sound. *Contemporary Music Review*, 22(4), 2003.

[33] Yoichi Nagashima. Biosensorfusion: New interfaces for interactive multimedia art. In *in Proceedings of International Computer Music Conference 1998*, 1998.

[34] K. Nishimoto, T. Maekawa, Y. Tada, K. Mase, and R. Nakatsu. Networked wearable musical instruments will bring a new musical culture. In *Proceedings of International Wearable Computer Symposium 2001*, Zurich, Switzerland, 2001.

[35] R. Post, M. Orth, E. Cooper, S. Strickon, J. Smith, and T. Machover. Musical jacket project. Available from: `http://www.media.mit.edu/hyperins/levis/` [cited June 2007].

[36] Miller Puckette. Combining event and signal processing in the max graphical programming enviornment. *Computer Music Journal*, 15(3):66–77, 1991.

[37] Miller Puckette. Pure data: another integrated computer music enviornment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa, Japan, 1996.

[38] Hal Rammel. Joe barrick's one-man band: A history of the piatarbajo and other one-man bands. *Musical Traditions*, (098), September 2002. Originally published in Musical Traditions No 8 Early 1990. Available from: `http://www.mustrad.org.uk/articles/barrick.htm` [cited May 2007].

[39] Pedro Rebelo. The haptic sensation and instrumental transgression. *Contemporary Music Review*, 25(1/2), Febuary/April 2006.

[40] John Richards. 32kg: Performance systems for a post-digital age. In *Proceedings of NIME-06*, Paris, France, 2006.

[41] Joel Ryan. Some remarks on musical instrument design at steim. *Contemporary Music Review*, 6(1):3–17, April 2007.

[42] David Z. Salz. The art of interaction: Interactivity, performativity, and computers. *The Journal of Aesthetics and Art Criticism*, 55(2):117–127, Spring 1997.

[43] Caleb Stuart. The object of performance: Aural performativity in contemporary laptop music. *Contemporary Music Review*, 22(4), 2003.

[44] Atau Tanaka. Musical performance practice on sensor-based instruments. In M. Wanderley and M. Battier, editors, *in Trends in Gestural Control of Music*, pages 389–405, Paris, France, 2000. Institut de Recherche et Coordination Acoustique MusiqueCentre Pompidou.

[45] Todd Winkler. Making motion musical: Gesture mapping strategies for interactive computer music. In *Proceedings of ICMC 1995*, Banff, Canada, September 1995.

[46] Charles K. Wolfe. The grand ole opry: When the skillet-lickers came to nashville. *Old Time Music*, (2):102, 1972. Reprinted from original newspaper.

List of Figures

A unit-daemon Source Code

Although this scope of this thesis does not include a detailed description and analysis of the software system, the following C++ source code is included to stimulate further development and document the programming experience gained by the author over the course of the project.

unit-daemon runs in Linux and requires the following libraries and applications: liblo (lightweight osc library), the Jack realtime audio daemon, Pure Data, aconnect, and alsaplayer with the Jack and text extensions.

- unit-daemon sources
  1. unit-daemon: main setup and loop
  2. Application: application handling class
  3. Button_Box: robotcowboy button_box class
  4. Config: configuration file class
  5. Devices: input device handling class
  6. Joystick_Device: Linux joystick class
  7. Osc_Server: Open Sound Control server class
  8. Playlist: circular play list class
  9. Serial_Device: serial device handling template class
  10. Sound_Feedback: sound file playback class
  11. Unit: core application handling class
- Configuration Files
  1. unit-daemon configuration
  2. play list
  3. joystick name to OSC address configuration
- unit-announce source
- unit-daemon udev rules

```
/*  main.cpp

    unit-daemon

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include <iostream>
#include "Unit.h"
#include "Devices.h"
#include "Button_Box.h"
#include "Playlist.h"
#include "Config.h"

using namespace std;

/* Server function declarations */
void setup_server(string port);

void server_error_callback(int num, const char *msg, const char *path);

int generic_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data);

int insert_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data);

int remove_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data);

int serial_device_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data);

int button_box_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data);

/* Signal exit handler */
void onExit(int nSig);

/* Classes */
Osc_Server server;
Devices input_devices;
Button_Box button_box;
Config config("config.txt");
Playlist playlist("playlist.txt");
Sound_Feedback sounds;

/* Terminal output */
ofstream filestr;

lo_address osc_send;    // osc send address to pd
bool done = false;      // is the loop running?

// mabey add commandline options
int main(int argc, char **argv)
{
```

```
    if(config.get("log") == "true")
    {
        // redirect cout to file
        streambuf *psbuf;
        filestr.open ("log.txt");

        psbuf = filestr.rdbuf();
        cout.rdbuf(psbuf);
    }

    cout << "******************" << endl;
    cout << "*** Unit Begin ***" << endl;
    cout << "******************" << endl;

    // load files
    cout << endl << "*** Load Config ***" << endl;
    if(config.load() < 0)
        return -1;

    cout << endl << "*** Load Playlist ***" << endl;
    if(playlist.load() < 0)
        return -1;

    // start server
    cout << endl << "*** OSC Server Begin ***" << endl;
    cout << "OSC Server Starting up ..." << endl;
    setup_server(config.get("server_port"));
    server.startListening();
    cout << "... Server running" << endl;

    // start jack
    Unit my_unit;
    cout << endl << "*** Start Jack ***" << endl;
    if(my_unit.startJack(config.get("jack_command")) < 0)
    {
        cout << "Jack failed to start ... exiting" << endl;
        return EXIT_FAILURE;
    }

    // salutation
    sleep(1);
    sounds.play(config.get("sound_folder")+"/you_are_robotcowboy.wav");

    // setup devices
    cout << endl << "*** Setup Devices ***" << endl;
    input_devices.config("joy_names.txt");
    input_devices.setupOSC((char *) config.get("send_addr").c_str(), (char *) config.get("send_port").c_str(),
"/pd/devices/");
    button_box.setBaud((char *) config.get("baud").c_str());
    button_box.setupOSC((char *) config.get("send_addr").c_str(), (char *) config.get("send_port").c_str(),
"/pd/transport");
    if(button_box.openDev((char *) config.get("button_box").c_str(), (char *) config.get("baud").c_str())  == 0)
        cout << "Button Box opened at " << config.get("button_box") << endl;
    input_devices.setup();

    if(config.get("debug") == "true")
    {
        button_box.printEvents(true);
        input_devices.printEvents(true);
    }

    // open pd with control patch
    cout << endl << "*** Start Pd ***" << endl;
    my_unit.startPd(config.get("puredata_command"));

    // open first patch
```

```
    sleep(3);
    osc_send = lo_address_new((char *) config.get("send_addr").c_str(), (char *) config.get("send_port").c_str
());
    lo_send(osc_send, "/pd/patch/open", "ss", (char *) playlist.file().c_str(), (char *) playlist.path().c_str
());

    // connect midi
    my_unit.aconnect("UA-25", "Pure Data");

    // ready sound
    sounds.play(config.get("sound_folder")+"/unit_ready.wav");

    // signal handling
    signal(SIGTERM, onExit);    // terminate
    signal(SIGQUIT, onExit);    // quit
    signal(SIGINT, onExit);     // interrupt

    // program main loop
    while (!done)
    {

        input_devices.listen();

        button_box.listen();

        // is there a load?
        if(button_box.check() > 0)
        {
            sounds.play(config.get("sound_folder")+"/bang.wav");

            lo_send(osc_send, "/pd/patch/close", "ss", (char *) playlist.file().c_str(), (char *) playlist.path
().c_str());
            playlist.next();

            lo_send(osc_send, "/pd/patch/open", "ss", (char *) playlist.file().c_str(), (char *) playlist.path
().c_str());
        }

        usleep(10); // debounce ... important!

    } // end main loop

    cout << endl << "*** Unit Shut Down ***" << endl;

    // end sound
    sounds.play(config.get("sound_folder")+"/unit_signing_off.wav");
    sleep(3);

    cout << endl << "Shutting down devices ..." << endl;
   // input_devices.cleanup();
    button_box.closeDev();
    cout << "... devices shut down" << endl;

    cout << endl << "Stopping Pure Data and Jack ..." << endl;
    my_unit.stopPd();
    cout << "... Pure Data stopped" << endl;
    my_unit.stopJack();
    cout << "... Jack stopped" << endl;

    cout << endl << "Stopping Osc Server ..." << endl;
    server.stopListening();
    cout << "... server stopped" << endl;

    // wait for all children just in case
    my_unit.cleanup();
```

```
    cout << endl << "*** Unit Sucessfully Shutdown ***" << endl;

    if(config.get("log") == "true")
        filestr.close();

    return EXIT_SUCCESS;
}

void setup_server(string port)
{
    server.setup(port.c_str(), server_error_callback);

    // add address handling callbacks
    server.addRecvMethod(NULL, NULL, generic_handler);
    server.addRecvMethod("/unit/device/sdl/start", "s", insert_handler);
    server.addRecvMethod("/unit/device/sdl/stop", "s", remove_handler);
    //server.addRecvMethod("/unit/device/serial", "s", serial_device_handler);
    server.addRecvMethod("/unit/button_box", "s", button_box_handler);
}

void server_error_callback(int num, const char *msg, const char *path)
{
    cout << "unit-daemon server error " << endl;//<< num << " in path "
       //  << path << ": " << msg << endl;
}

int generic_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data)
{
    cout << "Message recieved: " << path;

    for (int i = 0; i < argc; i++)
    {
        cout << " ";
        //printf("arg %d '%c' ", i, types[i]);
        lo_arg_pp((lo_type) types[i], argv[i]);
    }
    cout << endl;

    return 1;
}

/*  add and remove devices  */
int insert_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data)
{
    sounds.play(config.get("sound_folder")+"/oo.wav");
    input_devices.joyRegister((string) &argv[0]->s);
    //input_devices.printMap();

    return 1;
}

int remove_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data)
{
    sounds.play(config.get("sound_folder")+"/aw.wav");
    input_devices.joyUnregister((string) &argv[0]->s);
    //input_devices.printMap();

    return 1;
}

int serial_device_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data)
{
```

```
    return 1;
}

int button_box_handler(const char *path, const char *types, lo_arg **argv,
    int argc, void *data, void *user_data)
{
    unsigned char send = argv[0]->s;

    cout << "   sent \"" << send << "\"" << endl;

    button_box.send(&send, 1);

    return 1;
}

// handle exit signals from OS gracefully
void onExit(int nSig)
{
    done = true;
    cout << "   Signal Caught.  Exiting ..." << endl;
}
```

```
/*  Application.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef APPLICATION_H
#define APPLICATION_H

#include <unistd.h>
#include <iostream>
#include <fstream>
#include <string>
#include <signal.h>
#include <errno.h>

using namespace std;

/** \class  Application
    \brief  Application handling class

    Starts an app with a commandline, closes with signals, and checks running status
*/
class Application
{
    public:

        Application()   {};

        /**
            \brief  Constructor

            \param  cmdline_    cmdline string to launch app

            Ex: cmdline_ = "ls -l";
        */
        Application(string cmdline_);

        virtual ~Application();

        /**
            \brief  Launch the application

            runs the application cmdline specified in the constructor
            or setApp

            returns 0 on success or -1 on error
        */
        int launch();

        /**
            \brief  Send a Signal to the running application

            \param  signal  the signal to send (see signal.h)
                    i.e. SIGINT, SIGTERM, SIGQUIT, SIGWAIT, etc

            returns 0 on success and -1 on error
```

```
            Note: if application is not running, no signal will be
                  sent and sendSignal will return 0
        */
        int sendSignal(int signal);

        /**
            \brief  Get the status of the running application

            returns a char denoting the current status of the app:

            from the function:
            N   Process is not running

            from man ps:
            D    Uninterruptible sleep (usually IO)
            R    Running or runnable (on run queue)
            S    Interruptible sleep (waiting for an event to complete)
            T    Stopped, either by a job control signal or because it is being traced.
            W    paging (not valid since the 2.6.xx kernel)
            X    dead (should never be seen)
            Z    Defunct ("zombie") process, terminated but not reaped by its parent.

            returns -1 on error
        */
        char status();

        /**
            \brief  Set a new commandline

            \param  cmdline_    cmdline string to launch app

            Ex: cmdline_ = "ls -l";

            Note: DO NOT call this if the application is running,
            as the pid will be lost and you will lose control
            of the app
        */
        void setApp(string cmdline_);

        /**
            \brief  Get the pid of the application

            returns a pid > 0 if the app is running or 0 if not running
        */
        inline int getPid()     {return pid;}

        /**
            \brief  Get the applications name

            returns the name (first command line arg)
        */
        inline string getName() {return cmdline[0];}

    protected:

    private:

        // application info
        char *cmdline[20];    // cmdline args that launched app (20 args should be enough)
        pid_t pid;          // launched apps pid, 0 if not launched
};

#endif // APPLICATION_H
```

```
/*  Application.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Application.h"

Application::Application(string cmdline_)
{
    // strip cmdline string into a char array
    cmdline[0] = strtok((char *) cmdline_.c_str(), " ");
    for(int p = 1; p < 20 && (cmdline[p] = strtok(NULL, " ")) != NULL; p++);

    pid = 0;
}

Application::~Application()
{
    //dtor
}

/*  launches and application using the cmdline string
    return 0 on success or -1 on error */
int Application::launch()
{
    // virt fork off the parent
 pid_t pid_ = vfork();

    // bad pid, so exit with error
 if(pid_ < 0)
  return -1;  // error

 // parent
 if(pid_ > 0)
 {
     // fork worked
        pid = pid_;

     // check status just in case
        if(status() < 0)
        {
            pid = 0;
            return -1;
        }

  return 0;   // ok
 }

    // now running as child proc

    // launch the app with the cmdline
 execvp(cmdline[0], cmdline);

    return 0;   // ok
}
```

```
/* send a signal to the application (SIGTERM, SIGINT, ...)
    returns 0 on success or -1 on failure */
int Application::sendSignal(int signal)
{
    // valid pid?
    if(pid <= 0)
        return -1;  // error

    if(kill(pid, signal) != 0)
    {
        cout << "Application: '" << cmdline[0] << "' sendSignal failed: "
         << errno << " " << strerror(errno) << endl;
        return -1;  // error
    }

    return 0;   // ok
}

/*  get current status of the application
    returns status char on success and -1 on nonexisting process */
char Application::status()
{
    FILE *fpipe;
    char command[20];
    char line[100];

    sprintf(command, "ps %d", pid);

    if(!(fpipe = (FILE*) popen(command, "r")))
    {
        cout << "Application: ps failed to get status" << endl;
        return (char) -1;  // error
    }

    if(fgets(line, sizeof line, fpipe) == NULL)
        return (char) -1;
    if(fgets(line, sizeof line, fpipe) == NULL)
    {
        pid = 0;
        return 'N';   // no second line, so app must not be running
    }

    // strip out status char from ps stdout
    // second line, 3rd word, 1st char
    strtok(line, " "); // 1st word
    strtok(NULL, " "); // second word
    char *status = strtok(NULL, " ");

    return status[0];
}

/*  set a new commandline */
void Application::setApp(string cmdline_)
{
    // strip cmdline string into a char array
    cmdline[0] = strtok((char *) cmdline_.c_str(), " ");
    for(int p = 1; p < 20 && (cmdline[p] = strtok(NULL, " ")) != NULL; p++);

    pid = 0;
}
```

```
/*  Button_Box.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef BUTTON_BOX_H
#define BUTTON_BOX_H

#include <time.h>
#include <sys/time.h>
#include "Serial_Device.h"

/** \class  Button_Box
    \brief  button_box class inherited from Serial_Device */
class Button_Box: public Serial_Device
{
    public:

        Button_Box();

        virtual ~Button_Box();

        /** /brief  Checks for button events and sends cue

            Checks the buffer's first char for 'D' and 'U',
            keeps a timestamp between 'D' and 'U' events
            and sends an OSC message on a load:
            -   'load' timestmap >= 2
            -   'cue'  timestamp < 2

            returns 0 on a load and -1 on cue or nothing read

            Note:: setupOSC must be called to setup OSC addr info
        */
        int check();

        /**
            \brief  Print all of the events?

            \param  yesno   true = print all button events
        */
        void inline printEvents(bool yesno) {print_events = yesno;};

    protected:

    private:
        time_t timestamp;   // timestamp of last 'D' event
        bool print_events;  // printing control
};

#endif // BUTTON_BOX_H
```

```
/*  Button_Box.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Button_Box.h"

Button_Box::Button_Box()
{
    //timestamp = 0;
}

Button_Box::~Button_Box()
{
    //dtor
}

int Button_Box::check()
{
    // device is not open, so dont do anything
    if(dev_fd == -1)
        return 0;

    if(num_bytes > 0)  // read anything?
    {

        // box only sends 1 char
        if(buffer[0] == 'D')    // button down
        {
            time(&timestamp);

            if(print_events)   // print event info
                cout << "Button_Box: recieved \"" << buffer[0] << "\"" << endl;
        }

        else if(buffer[0] == 'U')   // button up
        {
            time_t now;
            time(&now);

            double t = difftime(now, timestamp);

            if(t >= 2)   // send a load event
            {
                cout << "Button_Box: Load!" << endl;
                return 1;
            }
            else    // send a cue event
            {
                if(lo_send(osc_server, addr, "s", "cue") == -1)
                        if(print_events)
                            cout << "OSC error" << lo_address_errno(osc_server)
                                 << lo_address_errstr(osc_server) << endl;
                cout << "Button_Box: Cue!" << endl;
            }
```

```
            if(print_events)   // print event info
                cout << "Button_Box: recieved \"" << buffer[0] << "\" " << t << endl;
        }
    }

    return 0; // nothing done or cue
}
```

```
/*  Config.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef CONFIG_H
#define CONFIG_H

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>

using namespace std;

/** \class  Config
    \brief  Class to read config file and retrieve config keys */
class Config
{
    public:

        /** \brief Constructor

            \param  filename    filename of the configfile to open, including path
        */
        Config(char *filename_);

        virtual ~Config();

        /**  \brief  Load configuration from file

            Ignores lines beginning with '#',
            format is 'key' 'value' with a
            space in between

            ex.
            # server address comment
            server_addr 127.0.0.1

            returns 0 on success or -1 if the file cannot be loaded (i.e. found)

            Note: very dumb, does not check for bad keys/vals so config file
            must be correct
        */
        int load();

        /** \brief  Loads joystick name mapping configuration

            Ignores lines beginning with '#',
            format is 'usb dev name' 'OSC device address' with a
            space in between

            ex.
            # saitek events sent to "/target address/saitek"
            "Saitek P990 Dual Analog Pad" saitek
```

```
            returns 0 on success or -1 if the file cannot be loaded (i.e. found)

            Note: very dumb, does not check for bad keys/vals so config file
            must be correct
        */
        int loadJoy();

        /** \brief  Get a configuration value

            \param  key key to fetch value for

            ex. get("server_addr"); returns the value for the server_addr

            returns value or "" (empty string) if key was not found

            Note: see configuration for keys / values

        */
        string get(char* key);

        /** \brief  Prints the current configuration keys and values */
        void print();

        /** \brief  Returns the config file filename */
        inline char *name() {return filename;};

    protected:

    private:

        char *filename;                     // Config filename
        map<string, string> config_map;    // map of config values
};

#endif // CONFIG_H
```

```cpp
/*  Config.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Config.h"

Config::Config(char *filename_)
{
    filename = filename_;
}

Config::~Config()
{
    //dtor
}

int Config::load()
{
    ifstream fin(filename, ios::in);

    if(!fin)    // open failed
    {
        cout << "Config: error opening file \"" << filename << "\"" << endl;
        return -1;
    }

    cout << "Config: loading " << filename << endl;

    string s;
    while(getline(fin, s))
    {
        if(s.size() >= 1 && s[0] == '#') {}
      //      cout << "   ignoring comment: " << s << endl;
        else if(s.size() >= 2)
        {
            istringstream ss(s);
            string key, val;
            ss >> key;
            ss.ignore();    // remove preceeding space
            getline(ss, val);

            cout << "   key: \"" << key << "\" val: \"" << val << "\"" << endl;
            config_map.insert(make_pair(key, val));
        }
    }

    cout << "Config: ready" << endl;
    fin.close();

    return 0;
}

int Config::loadJoy()
{
    ifstream fin(filename, ios::in);
```

```
    if(!fin)    // open failed
    {
        cout << "Config: error opening file \"" << filename << "\"" << endl;
        return -1;
    }

    cout << "Config: loading " << filename << endl;

    string s;
    while(getline(fin, s))
    {
        if(s.size() >= 1 && s[0] == '#') {}
      //      cout << "   ignoring comment: " << s << endl;
        else if(s.size() >= 2)
        {
            string key, val;

            key = s.substr(s.find_first_of("\"")+1, s.find_last_of("\"")-1);
            val = s.substr(s.find_last_of("\"")+2, s.size());

            cout << "   key: \"" << key << "\" val: \"" << val << "\"" << endl;
            config_map.insert(make_pair(key, val));
        }
    }

    cout << "Config: ready" << endl;
    fin.close();

    return 0;
}

string Config::get(char* key)
{
    return config_map[key];
}

void Config::print()
{
        int i = 0;

        map<string, string>::iterator c;
        for(c = config_map.begin(); c != config_map.end(); c++)
        {
            cout << i << " key: " << c->first << " val:  " << c->second << endl;
            i++;
        }
}
```

```
/*  Devices.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef DEVICES_H
#define DEVICES_H

#include <string>
#include <sstream>
#include <map>
#include <vector>

#include "Config.h"
#include "Joystick_Device.h"
#include "Serial_Device.h"

class Devices
{
    public:

        Devices();

        virtual ~Devices();

        /** \brief  Setup the OSC connection info

            \param  ip_ ip address to OSC server to send to, NULL sets localhost "127.0.0.1"
            \param  port_   port number of OSC server, ex. "4000"
            \param  osc_addr_    OSC address to send to, ex. "/test/serial/1"
        */
        void setupOSC(char *ip_, char *port_, char *osc_addr_);

        /** \brief  Opens all currently plugged in devices

            Called to init existing devices before starting event listening
        */
        void setup();

        /** \brief  Loads the OSC address config file
            \param  file    filename to the config file

            Ignores lines beginning with '#',
            format is 'usb dev name' 'OSC device address' with a
            space in between

            ex.
            # saitek events sent to "/target address/saitek"
            "Saitek P990 Dual Analog Pad" saitek

            returns 0 on success or -1 if the file cannot be loaded (i.e. found)

            Note: very dumb, does not check for bad keys/vals so config file
            must be correct
        */
        int config(char *file);
```

```
        /** \brief  Closes each joystick and removes it from the list */
        void cleanup();

        /** \brief  Toggles debug event output
            \param  yesno   true = prints all button, axes event information
        */
        void printEvents(bool yesno);

        /** \brief  Checks for and sends device events */
        void listen();

        /** \brief  Adds joystick to active list and opens it
            \param  dev name of the device, ie. '/dev/input/js0'
        */
        int joyRegister(string dev);

        /** \brief  Removes joystick to active list and closes it
            \param  dev name of the device, ie. '/dev/input/js0'
        */
        int joyUnregister(string dev);

        /** \brief  debug print of active device list */
        void printMap();

    protected:

    private:
        char *ip;       // ip to send device events to
        char *port;     // port
        char *osc_addr; // base OSC address to send to, device addr is concatenated
        map<string, Joystick_Device> joy_devices;   // active device list
        vector<Serial_Device> serial_devices;   // serial device list

        Config *name_mappings;  // device name -> OSC address config file
};

#endif // DEVICES_H
```

```
/*  Devices.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Devices.h"

Devices::Devices()
{
    ip = NULL;
    port = NULL;
    osc_addr = NULL;
    name_mappings = NULL;
}

Devices::~Devices()
{
    //dtor
}

void Devices::setupOSC(char *ip_, char *port_, char *osc_addr_)
{
    ip = ip_;
    port = port_;
    osc_addr = osc_addr_;
}

/*  Opens all exisiting linux joystick devices */
void Devices::setup()
{
    FILE *fpipe;
    char line[100];

    // call ls on the /dev/input dir to get available joysticks
    if(!(fpipe = (FILE*) popen("ls /dev/input | grep js*", "r")))
    {
        cout << "Devices: ls /dev/input failed run" << endl;
        return;  // error
    }

    while(fgets(line, sizeof line, fpipe) != NULL)
    {
        string temp = line;

        // grab joy name
        istringstream ss(temp);
        string dev;
        ss >> dev;
        joyRegister(dev);
    }

    return;
}

int Devices::config(char *file)
{
```

```
    name_mappings = new Config(file);

    // load the config file
    if(name_mappings->loadJoy() < 0)
        return -1;  // bad file

    return 0;
}

/*  Close each joystick and remove it from the map */
void Devices::cleanup()
{
    if(joy_devices.size() > 0)
    {
        map<string, Joystick_Device>::iterator c;
        for(c = joy_devices.begin(); c != joy_devices.end(); c++)
        {
            c->second.closeDev();
            joy_devices.erase(c);
        }
    }
}

/*  Print the joystick events yes/no> */
void Devices::printEvents(bool yesno)
{
    if(joy_devices.size() > 0)
    {
        // print joystick events (buttons, axes, etc)
        map<string, Joystick_Device>::iterator c;
        for(c = joy_devices.begin(); c != joy_devices.end(); c++)
            c->second.printEvents(yesno);
    }
}

/*  Listen for joy events and send osc */
void Devices::listen()
{
    if(joy_devices.size() > 0)
    {
        // check for events
        map<string, Joystick_Device>::iterator c;
        for(c = joy_devices.begin(); c != joy_devices.end(); c++)
            c->second.listen();
    }
}

int Devices::joyRegister(string dev)
{
    Joystick_Device new_device;

    // try opening the js
    if(new_device.openDev((char *) dev.c_str()) < 0)  // bad?
        return -1;  // bad!

    string addr;

    // assign osc addr based on name or device name
    if(name_mappings != NULL)
    {
        string name = name_mappings->get(new_device.name());

        cout << "[" << name << "]" << endl;

        if(name != ""){
             cout << "inside" << endl;
```

```
            addr = osc_addr + name;}
        else
        addr = osc_addr + dev;
    }
    else
        addr = osc_addr + dev;

    // setup osc
    new_device.setupOSC(ip, port, (char *) addr.c_str());

    cout << "Joystick registered: \"" << dev << "\"" << endl;
    new_device.printInfo();
    joy_devices.insert(make_pair(dev, new_device));

    return 0;
}

int Devices::joyUnregister(string dev)
{
    // no devices?
    if(joy_devices.size() == 0)
    {
        cout << "Joystick map is empty so \"" << dev << "\" cannot be unregistered" << endl;
        return 0;
    }

    map<string, Joystick_Device>::iterator c = joy_devices.find(dev);

    // didnt find it?
    if(c == joy_devices.end())
    {
        cout << "Joystick \"" << dev << "\" was not found, so it cannot be unregistered" << endl;
        return 0;
    }

    c->second.closeDev();
    joy_devices.erase(c);

    cout << "Joystick \"" << dev << "\" has been unregistered" << endl;

    return 0;
}

void Devices::printMap()
{
    int i = 0;

    map<string, Joystick_Device>::iterator c;
    for(c = joy_devices.begin(); c != joy_devices.end(); c++)
    {
        cout << i << " key: " << c->first << " val:  " << c->second.name() << endl;
        i++;
    }
}
```

```
/*  Joystick_Devices.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef JOYSTICK_DEVICE_H
#define JOYSTICK_DEVICE_H

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

#include <iostream>
#include <string>

#include <linux/input.h>
#include <linux/joystick.h>
#include <lo/lo.h>

using namespace std;

/** \class  Joystick_Device
    \brief  Handles a Joystick device

    Uses the Linux joystick event system to open, read, and close a joystick
*/
class Joystick_Device
{
     public:

        Joystick_Device();

        ~Joystick_Device();

        /**
            \brief  Open linux joystick device

            \param  dev device name aka "/dev/input/js0"

            returns 0 on success and -1 on failure
        */
        int openDev(char *dev);

        /**
            \brief  Close joystick device
        */
        void closeDev();
```

```
    /** \brief  Setup the OSC connection info

        \param  ip_ ip address to OSC server to send to, NULL sets localhost "127.0.0.1"
        \param  port_   port number of OSC server, ex. "4000"
        \param  osc_addr    OSC address to send to, ex. "/test/serial/1"
    */
    void setupOSC(char *ip_, char *port_, char *osc_addr);

    /**
        \brief  Handles device events and sends correpsonding OSC messages

        Call this inside a loop, does not block, does nothing if joy has not been opened
    */
    void listen();

    /**
        \brief  Print device info
    */
    void printInfo();

    /**
        \brief  Get device name i.e. "/dev/input/js0"
    */
    char inline *devName()     {return dev_name;}

    /**
        \brief  Get joystick name
    */
    char inline *name()     {return js_name;}

    /**
        \brief  Get number of Axes
    */
    int inline numAxes()    {return num_axes;}

    /**
        \brief  Get number of buttons
    */
    int inline numButtons() {return num_buttons;}

    /**
        \brief  Print all of the events?

        \param  yesno   true = print all device events (buttons, axes, etc)
    */
    void inline printEvents(bool yesno) {print_events = yesno;};

    /** \brief  Returns true if device is open */
    inline bool isOpen()    {if(dev_fd > -1) return true; else return false;}

protected:

private:

// linux joystick info
char *dev_name;
char *js_name;
int dev_fd;
int num_axes;
int num_buttons;
fd_set set;     // set for select
timeval tv;     // timeout for select
js_event event; // joystick event struct

// osc connection info
lo_address osc_server;
```

```
    char *ip;
    char *port;
    char *addr;
    string axis_addr;
    string button_addr;

    // printing control
    bool print_events;
};

#endif // JOYSTICK_DEVICE_H
```

```
/*  Joystick_Device.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Joystick_Device.h"

Joystick_Device::Joystick_Device()
{
    // linux joystick info
    dev_name = NULL;
    js_name = NULL;
    dev_fd = -1;
    num_axes = 0;
    num_buttons = 0;

    // setup timeouts
    tv.tv_sec = 1;
    tv.tv_usec = 0;

    // osc connection info
    osc_server = NULL;
    ip = "";
    port = "";
    addr = "";
    axis_addr = "";
    button_addr = "";

    // printing control
    print_events = false;
}

Joystick_Device::~Joystick_Device()
{
    //dtor
}

/*  open device with linux joystick dev name aka /dev/js0
    returns 0 on success and -1 on failure */
int Joystick_Device::openDev(char *dev)
{
    // open the dev
    string dev_path = "/dev/input/" + (string) dev;
    if((dev_fd = open((char *) dev_path.c_str(), O_RDONLY)) < 0)
    {
        cout << "Joystick_Device: Bad joystick device: "
        << dev << " " << strerror(errno) << endl;
        return -1;
    }

    dev_name = dev;

    // set nonblocking
    fcntl(dev_fd, F_SETFL, O_NONBLOCK);

    // query for device info
```

```
    int version;
    ioctl(dev_fd, JSIOCGVERSION, &version);
    // exit if using old (non-event) joystick api
    if(version < 0x010000)
    {
        cout << "Joystick_Device: driver for " << dev_name
             << " uses old joy device version < 1.0" << endl;
        return -1;
    }
 ioctl(dev_fd, JSIOCGAXES, (int) &num_axes);
 ioctl(dev_fd, JSIOCGBUTTONS, (int) &num_buttons);

    js_name = new char[128];
 ioctl(dev_fd, JSIOCGNAME(128), js_name);

    return 0;
}

/*  close device */
void Joystick_Device::closeDev()
{
    // close joystick
    close(dev_fd);

    // free addr
    lo_address_free(osc_server);

    // reinit vals incase we want to reuse this object
    js_name = NULL;
    dev_fd = -1;
    num_axes = 0;
    num_buttons = 0;

    // osc connection info
    osc_server = NULL;
    ip = "";
    port = "";
    addr = "";
    axis_addr = "";
    button_addr = "";
}

/* setup the OSC connection info */
void Joystick_Device::setupOSC(char *ip_, char *port_, char *osc_addr)
{
    ip = ip_;
    port = port_;
    addr = osc_addr;

    // setup osc send address
    osc_server = lo_address_new(ip, port);

    // set addr
    axis_addr   =   addr + (string) "/axis";
    button_addr =   addr + (string) "/button";
}

/*  handles device events and sends correpsonding OSC messages
    call this inside a loop, is nonblocking */
void Joystick_Device::listen()
{
    // device is not open, so dont do anything
    if(dev_fd == -1)
        return;

    FD_ZERO(&set);
```

```
    FD_SET(dev_fd, &set);

    if(!select(dev_fd+1, &set, NULL, NULL, &tv))
        return; // nothing read

    if(read(dev_fd, &event, sizeof(event)) != sizeof(event))    // bad read
    {
        dev_fd = -1; // mark as bad so no more reading
        if(errno == ENODEV) return;  // exit if device has already been unplugged (no device)
        cout << "Joystick \"" << dev_name << "\" read error: " << strerror(errno) << endl;
        return;
    }

    if(print_events)    // debug printing
        cout << " Joy: " << dev_name
             << " Button: " << (int) event.number
             << " State: " << (int) event.value << endl;

    // check for messages
    switch (event.type)
    {

        case 1: // buttons
        {
            if(lo_send(osc_server, button_addr.c_str(), "ii", event.number, event.value) == -1)
                if(print_events)
                    cout << "OSC error" << lo_address_errno(osc_server)
                         << lo_address_errstr(osc_server) << endl;
            break;
        }

        case 2: // axes
        {
            if(lo_send(osc_server, axis_addr.c_str(), "ii", event.number, event.value) == -1)
                if(print_events)
                    cout << "OSC error" << lo_address_errno(osc_server)
                         << lo_address_errstr(osc_server) << endl;
            break;
        }
    } // end switch

    return; //ok
}

/*  print device info */
void Joystick_Device::printInfo()
{
    cout << "OSC Addr: "  << addr << endl;
    cout << "  Dev Name: " << dev_name << endl;
    cout << "  Name: " << js_name << endl;
    cout << "  Num Axes: " << num_axes<< endl;
    cout << "  Num Buttons: " << num_buttons << endl;
}
```

```
/*  Osc_Server.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef OSC_SERVER_H
#define OSC_SERVER_H

#include "lo/lo.h"  // liblo

/** \class  Osc_Server
    \brief  Osc server class

    Starts, stops, and assigns callback functions
*/
class Osc_Server
{
    public:

        Osc_Server();

        ~Osc_Server();

        /**
            \brief  Setup the osc server

            \param  port    The port number to recieve on
            \param  error   function pointer to a method to handle recieve errors
        */
        void setup(const char *port, void (*error)(int num, const char *msg, const char *where));

        /*
            \brief  Sets the server error callback function

            \param  error   function pointer to a method to handle recieve errors

            Note: must be called *before* calling setRecv
        */
      //  void setErrorCallback(void (*error)(int num, const char *msg, const char *where));

        /**
            \brief  Sets an OSC path handling callback

            \param  path    The OSC path the callback will handle
            \param  type    The types of the data items in the message
            \param  method  function pointer to a method to handle the OSC message with path and type

            This method will be called whenever a message is recieved at OSC address "path" and contains
arguments or "type".
        */
        void addRecvMethod(const char *path, const char *types,
                           int (*method)(const char *path, const char *types, lo_arg **argv, int argc, lo_message
msg, void *user_data));

        /**
            \brief  Removes a OSC path handling callback
```

```
            \param  path    The OSC path the callback will handle
            \param  type    The types of the data items in the message

            Removes the callback method set for OSC address "path" and arguments of "type".
        */
        void delRecvMethod(const char *path, const char *types);

        /**
            \brief  Starts the listening server

            When running, the server will call any callback methods set using addRecvMethod.
        */
        void startListening();

        /**
            \brief Stops the listening server
        */
        void stopListening();


    protected:
    private:
        lo_server_thread recv_thread;   // osc recieve thread
    //    void (*error_callback)(int num, const char *msg, const char *where);    // function pointer to error
callback
};

#endif
```

```
/*  Osc_Server.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Osc_Server.h"

// CONSTRUCTOR / DESTRUCTOR

Osc_Server::Osc_Server()
{
    recv_thread = NULL;
}

Osc_Server::~Osc_Server()
{
    // free mem
    if(recv_thread != NULL)   lo_server_thread_free(recv_thread);
}

// SET PORT

void Osc_Server::setup(const char *port, void (*error)(int num, const char *msg, const char *where))
{
    // create new recv thread with port num
    recv_thread = lo_server_thread_new(port, error);
}

// SERVER CALLBACKS
/*
void Osc_Server::setErrorCallback()
{
    error_callback = error;
}
*/
void Osc_Server::addRecvMethod(const char *path, const char *types,
                        int (*method)(const char *path, const char *types, lo_arg **argv, int argc, lo_message
msg, void *user_data))
{
    lo_server_thread_add_method(recv_thread, path, types, method, NULL);
}

void Osc_Server::delRecvMethod(const char *path, const char *types)
{
    lo_server_del_method(recv_thread, path, types);
}

// SERVER CONTROL

void Osc_Server::startListening()
{
        lo_server_thread_start(recv_thread);
}

void Osc_Server::stopListening()
{
```

```
        lo_server_thread_stop(recv_thread);
}
```

```
/*  Playlist.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef PLAYLIST_H
#define PLAYLIST_H

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

/** \class  Playlist
    \brief  Loads and handles a circular play list of filenames
*/
class Playlist
{
    public:

        /** \brief  Constructor

            \param  filename of the playlist file to open, including path
        */
        Playlist(char *filename_);

        virtual ~Playlist();

        /** \brief  Load the playlist

            Ignores lines beginning with '#',
            format is one file per line, including full path

            ex.
            # this is a comment
            /home/user/awesomesong.pd

            returns 0 on success or -1 if the file cannot be loaded (i.e. found)

            Note: very dumb, just reads lines, so watch the whitespace in the playlist file
        */
        int load();

        /**  \brief  Return the current song filename

            Includes fullpath: /home/user/awesomesong.pd
        */
        inline string song() {return *pos;};

        /** \brief  Returns the current song filename

            Does not include path: awesomesong.pd
        */
        string file();
```

```
        /** \brief  Returns the current song path

            Does not include filename: /home/user/
        */
        string path();

        /** \brief  Move to the next song in the playlist */
        void next();

        /** \brief  Move to the previous song in the playlist */
        void prev();

        /** \brief  Print the playlist */
        void print();

        /** \brief  Returns the playlist file filename */
        inline char *name() {return filename;};

    protected:

    private:

        char *filename;                 // playlist filename
        vector<string> list;            // string vector of playlist items
        vector<string>::iterator pos;   // current position in the playlist
};

#endif // PLAYLIST_H
```

```
/*  Playlist.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Playlist.h"

Playlist::Playlist(char *filename_)
{
    filename = filename_;
    pos = list.begin();
}

Playlist::~Playlist()
{
    //dtor
}

int Playlist::load()
{
    ifstream fin(filename, ios::in);

    if(!fin)    // open failed
    {
        cout << "Playlist: error opening file \"" << filename << "\"" << endl;
        return -1;
    }

    cout << "Playlist: loading " << filename << endl;

    string s;
    int i = 0;
    while(getline(fin, s))
    {
        if(s.size() >= 1 && s[0] == '#') {}
      //      cout << "   ignoring comment: " << s << endl;
        else if(s.size() >= 2)
        {

                cout << "   Path " << i << " " << s << endl;
                list.push_back(s);
                i++;
        }
    }

    cout << "Playlist: ready" << endl;
    pos = list.begin();
    fin.close();

    return 0;
}

string Playlist::file()
{
    int loc = pos->find_last_of("/") + 1;
```

```
    return pos->substr(loc, pos->size());
}

string Playlist::path()
{
    int loc = pos->find_last_of("/");

    return pos->substr(0, loc);
}

void Playlist::next()
{
    pos++;
    if(pos == list.end())   pos = list.begin();
}

void Playlist::prev()
{
    if(pos == list.begin())
        pos = list.end()-1;
    else
        pos--;
}

void Playlist::print()
{
        int i = 0;
        for(vector<string>::iterator c = list.begin(); c < list.end(); c++)
        {
            cout << i << " " << *c << endl;
            i++;
        }
}
```

```
/*  Serial_Device.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef SERIAL_DEVICE_H
#define SERIAL_DEVICE_H

#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>
#include <iostream>
#include <string>
#include <errno.h>
#include <lo/lo.h>

using namespace std;

/** \class Serial_Device
    Template class for serial device handling
*/
class Serial_Device
{
    public:

        Serial_Device();

        ~Serial_Device();

        /** \brief  Open serial port with the dev name and speed

            \param  dev name of serial port device to open, ex. "/dev/ttyS0"
            \param  baud    baud speed of the device (see setBaud) ex. "9600"

            returns 0 on success and -1 on failure
        */
        int openDev(char *dev, char *baud);

        /** \brief  Close serial port */
        void closeDev();

        /** \brief  Setup the OSC connection info

            \param  ip_ ip address to OSC server to send to, NULL sets localhost "127.0.0.1"
            \param  port_   port number of OSC server, ex. "4000"
            \param  osc_addr    OSC address to send to, ex. "/test/serial/1"
        */
        void setupOSC(char *ip_, char *port_, char *osc_addr);

        /** \brief  Listen for incoming bytes and null terminates buffer

            returns number of bytes read on success or -1 on read error,
            will return 0 if no data to be read
            Note: nonblocking function using select, so call it within a loop
        */
```

```
        int listen();

        /** \brief  Send some bytes from a char device

            \param  send_chars  char buffer to send
            \param  n_bytes number of bytes to send from send_chars buffer

            returns number of bytes read or -1 on error

            Note: returns 0 if device is not open
        */
        int send(unsigned char *send_chars, int n_bytes);

        /** \brief  Set the speed of the serial device

            \param  baud    baud speed of the device:
                            50     50 baud
                            75     75 baud
                            110     110 baud
                            134     134.5 baud
                            150     150 baud
                            200     200 baud
                            300     300 baud
                            600     600 baud
                            1200 1200 baud
                            1800 1800 baud
                            2400 2400 baud
                            4800 4800 baud
                            9600 9600 baud
                            19200 19200 baud
                            38400 38400 baud
                            57600 57,600 baud
                            115200 115,200 baud

        */
        int setBaud(char *baud);

        /** \brief  Get pointer to char buffer

            returns pointer to serial device input buffer
        */
        inline unsigned char *getBuffer()    {return buffer;};

        /** \brief  Returns true if device is open */
        inline bool isOpen()    {if(dev_fd > -1) return true; else return false;}

    protected:

    // serial device info
    char *dev_name;         // port name of the serial device
    int dev_fd;             // serial device file descriptor
    unsigned char *buffer;  // Input buffer
    int  num_bytes;         // Number of bytes read
    fd_set set;     // set for select
    timeval tv;     // timeout for select

    // osc connection info
    lo_address osc_server;
    char *ip;
    char *port;
    char *addr;

    private:
};

#endif // SERIAL_DEVICE_H
```

```cpp
/*  Serial_Device.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Serial_Device.h"

Serial_Device::Serial_Device()
{
    // serial device info
    dev_name = NULL;
    dev_fd = -1;
    buffer = new unsigned char[255];
    num_bytes = 0;

    // setup timeouts
    tv.tv_sec = 1;
    tv.tv_usec = 0;

    // osc connection info
    osc_server = NULL;
    ip = NULL;
    port = NULL;
    addr = NULL;
}

Serial_Device::~Serial_Device()
{
    //dtor
}

/*  open serial port with the dev name and speed
    returns 0 on success and -1 on failure */
int Serial_Device::openDev(char *dev, char *baud)
{
    // open port : read/write | not controlling | non blocking | ignore DCD state
    if((dev_fd = open(dev, O_RDWR | O_NOCTTY | O_NONBLOCK |  O_NDELAY)) == -1)
    {
        // Could not open the port.
        cout << "Serial_Device: Unable to open \"" << dev << "\": " << strerror(errno) <<  endl;
        return -1;  // error
    }

    fcntl(dev_fd, F_SETFL, FNONBLOCK);  // set nonblocking

    setBaud(baud);   // set speed

    dev_name = dev;

    return 0;   // ok
}

/*  close serial device */
void Serial_Device::closeDev()
{
    // close serial port
```

```
    close(dev_fd);

    // free addr
    lo_address_free(osc_server);

    // reinit vals incase we want to reuse this object
    dev_fd = -1;

    // osc connection info
    /*
    osc_server = NULL;
    ip = NULL;
    port = NULL;
    addr = NULL;
    */
}

/* setup the OSC connection info */
void Serial_Device::setupOSC(char *ip_, char *port_, char *osc_addr)
{
    ip = ip_;
    port = port_;
    addr = osc_addr;

    // setup osc send address
    osc_server = lo_address_new(ip, port);
}

/*  listen for incoming bytes and null terminates buffer
    returns number of bytes read

    Note: nonblicking function*/
int Serial_Device::listen()
{
    // device is not open, so dont do anything
    if(dev_fd == -1)
        return 0;

    FD_ZERO(&set);
    FD_SET(dev_fd, &set);

    if(!select(dev_fd+1, &set, NULL, NULL, &tv))
        return 0; // nothing read

    // read is nonblocking
    if((num_bytes = read(dev_fd, buffer, sizeof(buffer))) < 0)
    {
        cout << "Serial Device \"" << dev_name << "\" read error: " << strerror(errno) << endl;
        return -1;
    }

    // null terminate for a cstring
    buffer[num_bytes] = '\0';

    return num_bytes;   // something read
}

/*  send some bytes from a char device
    returns number of bytes read or -1 on error */
int Serial_Device::send(unsigned char *send_chars, int n_bytes)
{
    int n = 0;

    if((n = write(dev_fd, send_chars, n_bytes)) == -1)
    {
        cout << "Serial Device \"" << dev_name << "\" send error: " << strerror(errno) << endl;
```

```
    }
    return n;
}

/*  set the baud rate of the serial Device
    returns 0 on success or -1 on error */
int Serial_Device::setBaud(char *baud)
{
    struct termios options;

    string b = baud;
    int speed = -1;

    string speeds[18] = {"50", "75", "110", "134", "150", "200", "300", "600", "1200", "1800",
                "2400", "4800", "9600", "19200", "38400", "57600", "115200"};
    int bauds[18] = {B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400,
                B4800, B9600, B19200, B38400, B57600, B115200};

    // check for valid baud
    for(int i = 0; i < 18; i++)
    {
        if(b == speeds[i])
        {
            speed = bauds[i];
            break;
        }
    }

    if(speed == -1) // bad baud
    cout << "Serial_Device: bad baud rate \"" << baud << "\"" << endl;
    return -1;

    // Get the current options for the port...
    tcgetattr(dev_fd, &options);

    // Set the baud rates
    cfsetispeed(&options, speed);
    cfsetospeed(&options, speed);

    // Enable the receiver and set local mode..
    options.c_cflag |= (CLOCAL | CREAD | CS8);

    // No parity (8N1)
    options.c_cflag &= ~PARENB;
    options.c_cflag &= ~CSTOPB;
    options.c_cflag &= ~CSIZE;  // important to set size last

    // No hardware flow control
    options.c_cflag &= ~CRTSCTS;

    //Set the new options for the port...
    tcsetattr(dev_fd, TCSANOW, &options);

    return 0; // ok
}
```

```
/*  Sound_Feedback.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef SOUND_FEEDBACK_H
#define SOUND_FEEDBACK_H

#include <string>
#include "Application.h"

/** \class  Sound_Feedback
    \brief  Plays soundfiles

    Calls alsaplayer with jack output (alsaplayer -i text -o jack)
    to play sound files
*/
class Sound_Feedback
{
    public:

        Sound_Feedback();

        virtual ~Sound_Feedback();

        /**
            \brief  Plays a soundfile using alsaplayer
            \param  filename    name of the soundfile to play
        */
        int play(string filename);

    protected:

    private:
        Application sound_player;
};



#endif // SOUND_FEEDBACK_H
```

```
/*  Sound_Feedback.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Sound_Feedback.h"

Sound_Feedback::Sound_Feedback()
{
    //ctor
}

Sound_Feedback::~Sound_Feedback()
{
    //dtor
}

int Sound_Feedback::play(string filename)
{
    //cout << "alsaplayer -i text -o jack " + filename << endl;
    sound_player.setApp("alsaplayer -i text -o jack " + filename);
    sound_player.launch();

    return 0;
}
```

```
/*  Unit.h

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#ifndef UNIT_H
#define UNIT_H

#include <string>
#include <iostream>
#include <sstream>
#include <signal.h>
#include <vector>
#include <sys/types.h>
#include <sys/wait.h>
#include <lo/lo.h>
#include <SDL/SDL.h>
#include <time.h>
#include <sys/time.h>

#include "Application.h"
#include "Osc_Server.h"
#include "SDL_Device.h"
#include "Serial_Device.h"
#include "Sound_Feedback.h"

/** \class  Unit
    \brief  Controls main applications

    Starts and stops Jack and Pure Data, calls aconnect to connect alsa MIDI devices
*/
class Unit
{
    public:

        Unit();

        virtual ~Unit();

        /**
            \brief  Starts the Jack realtime audio daemon
            \param  cmd commandline string to start Jack,
            i.e. 'jackd -R -p128 -dalsa -dhw:1 -r44100 -p512 -n3 -S'
        */
        int startJack(string cmd);

        /**
            \brief  Starts Pure Data
            \param  cmd commandline string to start Pd, i.e. 'pd -alsamidi -jack'
        */
        int startPd(string song);

        /**
            \brief  Calls aconnect to connect alsa MIDI devices
            \param  outport MIDI output to connect
            \param  inport  MIDI input to connect
```

```
            You can discover the names by running 'aconnect -lio'
        */
        int aconnect(string outport, string inport);

        /**
            \brief  Stops Jack

            Tries to shutdown Jack nicely by sending it a SIGQUIT, but will SIGKILL it
            after 3 secs if it hangs (this can be a problem every now and then)
        */
        int stopJack();

        /**
            \brief  Stops Pure Data

            Shutdowns Pd nicely by sending it a SIGINT
        */
        int stopPd();

        /**
            \brief  Waits for any child processes

            A waitpid wrapper
        */
        void cleanup();

    protected:

    private:

        Application jack;
        Application pd;
};

#endif // UNIT_H
```

```cpp
/*  Unit.cpp

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include "Unit.h"

Unit::Unit()
{
    //ctor
}

Unit::~Unit()
{
    //dtor
}

int Unit::startJack(string cmd)
{
    jack.setApp(cmd);
    jack.launch();

    // not running?
    while(jack.status() == 'N')
        return -1;

    return 0;
}

int Unit::startPd(string cmd)
{
    pd.setApp(cmd);
    pd.launch();

    sleep(1);

    // not running?
    if(pd.status() == 'N')
        return -1;

    return 0;
}

int Unit::aconnect(string outport, string inport)
{
    FILE *fpipe;
    char line[100];

    if(!(fpipe = (FILE*) popen("aconnect -lio", "r")))
    {
        cout << "Unit: aconnect -lio failed run" << endl;
        return -1;  // error
    }

    int in_id = -1, out_id = -1;
```

```
    while(fgets(line, sizeof line, fpipe) != NULL)
    {
        string temp = line;

        if(temp.find(inport) != string::npos && temp.find("client") != string::npos)
        {

            istringstream ss(temp);
            string ignore;

            ss >> ignore >> in_id;
            cout << "   Midi inport  : " << inport << " id: " << in_id << endl;
        }

        if(temp.find(outport) != string::npos && temp.find("client") != string::npos)
        {

            istringstream ss(temp);
            string ignore;

            ss >> ignore >> out_id;
            cout << "   Midi outport : " << outport << " id: " << out_id << endl;;
        }
    }

    if(in_id == -1 || out_id == -1)
    {
        cout << "Unit: aconnect could not connect " << outport << " to " << inport << endl;
        return -1;
    }

    ostringstream cmd;
    cmd << "aconnect " << out_id << " " << in_id;
    cout << cmd.str() << endl;

    Application aconnect(cmd.str());
    aconnect.launch();

    return 0;
}

int Unit::stopJack()
{
    jack.sendSignal(SIGQUIT);

    time_t timestamp;
    time_t now;
    double t;
    time(&timestamp);

    // not running?
    while(jack.status() == 'N')
    {
        time(&now);
        if((t = difftime(now, timestamp)) > 2)  // 3 sec timeout
        {
            // if jack stalls due to soundcard "in use", then kill it mean likes
            jack.sendSignal(SIGKILL);
            cout << "Unit: jackd hung, so I had to kill it" << endl;
            return -1;
        }
    }

/*
    sleep(1);
```

```
    // if jack stalls due to soundcard "in use", then kill it mean likes
    if(jack.status() != 'N')
    {
        jack.sendSignal(SIGKILL);
        cout << "Unit: jackd hung, so I had to kill it" << endl;
    }
*/
    return 0;
}

int Unit::stopPd()
{
    pd.sendSignal(SIGINT);

    return 0;
}

void Unit::cleanup()
{
    if(waitpid(-1, NULL, 0) == -1)
        cout << "Wait error" << endl;
}
```

```
#
# config text file
# '#' lines are comments
#
# format: key value
# must be a space between!

# server stuff
server_addr 127.0.0.1
server_port 7770

# send osc stuff
send_addr 127.0.0.1
send_port 8880

# button box dev name
button_box /dev/ttyUSB0
baud 9600

# sound folder (no trailing slash)
sound_folder sounds

# don't use the realtime option on both jack and pd at the same time,
# it can cause problems

# jack commandline
jack_command jackd -R -p128 -dalsa -dhw:1 -r44100 -p512 -n3 -S

# pd commandline with control patch
puredata_command pd -alsamidi -jack /home/dano/Creative/pd/Unit-Control.pd

# log? - aka print all output to a log file, otherwise to stdout
log false

# debug - aka print lots of events
debug false
```

```
#
# playlist text file
#
# '#' lines are comments
#

/home/dano/Creative/pd/BimBom.pd
/home/dano/Creative/pd/RunningMan.pd
```

```
#
# joystick name mappings
# '#' lines are comments
#
# format: key value
# must be a space between!

"LuenKeung Co.,Ltd USB  Joystick" ps2black

"Saitek P990 Dual Analog Pad" saitek

"WiseGroup.,Ltd MP-8866 Dual USB Joypad" ps2purple
```

```
/*  main.cpp

    unit-announce

  Copyright (C) 2007 Dan Wilcox

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

*/
#include <iostream>
#include <unistd.h>
#include <string>

#include "lo/lo.h"

using namespace std;

int main(int argc, char *argv[])
{
        if(argc < 4)
        {
            cout << "Usage: unit-announce <osc_dest_addr> <osc_dest_port> <device_name>" << endl << endl;
            exit(EXIT_FAILURE);
        }

        string addr = (string) argv[1];
        string port = (string) argv[2];
        string device = (string) argv[3];

        // check address arg
        for(int c = 0; c < (int) addr.length(); c++)
        {
            if(!isgraph((char) addr.at(c)))
            {
                cout << "Invalid addr name \"" << addr << "\"" << endl;
                exit(EXIT_FAILURE);
            }
        }

        // check port arg
        for(int c = 0; c < (int) port.length(); c++)
        {
            if(!isdigit((char) port.at(c)))
            {
                cout << "Invalid port number \"" << port << "\"" << endl;
                exit(EXIT_FAILURE);
            }
        }

        // check device name arg
        for(int c = 0; c < (int) port.length(); c++)
        {
            if(!isalnum((char) port.at(c)))
            {
                cout << "Invalid device name \"" << device << "\"" << endl;
                exit(EXIT_FAILURE);
            }
        }
```

```
        // send to localhost
        lo_address t = lo_address_new(NULL, (char *) port.c_str());

        if(lo_send(t, (char *) addr.c_str(), "s", (char *) device.c_str()) == -1)
        {
            cout << "OSC error " << lo_address_errno(t) << " " << lo_address_errstr(t) << endl;
            exit(EXIT_FAILURE);
        }

        cout << "Sent: <" << addr << ", "<< device << "> at port " << port << endl;

        return 0;
}
```

```
#
# unit-daemon udev rules
#
# launches unit-announce to send osc notification of device insert/removal to unit-daemon
#
# "add" actions are device insertions and device attributes are used to match the device
#
# "remove" actions are matched using ENV variables since the SYSFS node for the device is gone
# and thus the attributes have been deleted
#
# rules built using:
# - udevinfo -a -p $(udevinfo -q path -n /dev/*device*) : attributes
# - sudo udevmonitor --env /dev/*device* : events and env vars
#
#############################################################################################

#############################################################
# joystick devices
#
KERNEL=="js[0-9]*", ACTION=="add", RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce
/unit/device/sdl/start 7770 %k"

KERNEL=="js[0-9]*", ACTION=="remove", RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce
/unit/device/sdl/stop 7770 %k"

#############################################################
# USB tty serial devices
#
KERNEL=="ttyUSB[0-9]*", ACTION=="add", SUBSYSTEMS=="tty",
RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce /unit/device/start 7770 %k"

KERNEL=="ttyUSB[0-9]*", ACTION=="remove", SUBSYSTEMS=="tty",
RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce /unit/device/stop 7770 %k"

#############################################################
# Video 4 Linux devices aka webcams, etc
#
KERNEL=="video[0-0]*", ACTION=="add", SUBSYTEM=="video4linux",
RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce /unit/device/start 7770 %k"

KERNEL=="video[0-9]*", ACTION=="remove", SUBSYSTEM=="video4linux",
RUN="/home/dano/thesis/src/unit-announce/bin/Release/unit-announce /unit/device/stop 7770 %k"
```