# Responsive Sound Surfaces

by
Michael Daniel Wu

B.A., Computer-Music
Vassar College, 1990

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Media Arts and Sciences

at the Massachusetts Institute of Technology
September, 1994



Author_____
Program in Media Arts and Sciences
August 5, 1994

Certified by_____
A
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by_____
Departme...
Program in Media Arts and Sciences

# DISCLAIMER NOTICE

The accompanying media item for this thesis is available in the MIT Libraries or Institute Archives.

Thank you.

# Responsive Sound Surfaces

by
Michael Daniel Wu

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on August 5, 1994 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Media Arts and Sciences

## Abstract

Responsive sound surfaces are material surfaces, embedded with proximity sensors, that react with sound to human movement. They create a dynamic relationship between movement, space, and sound that engages a participant in a creative endeavor. Responsive sound surfaces are presented and developed as one possible model in the relatively unexplored area of participatory interactive systems. This work identified, designed, and implemented an initial set of software components necessary to realize interactive experiences within the context of responsive sound surfaces. In addition, a framework for organizing cooperating software components was designed and various approaches to structuring non-linear, unfolding interactive experiences were explored. A video submitted with this thesis demonstrates some of my results.

Thesis Supervisor: Tod Machover
Associate Professor of Music and Media

This work was supported in part by Sega of America and Yamaha Corporation.

# Responsive Sound Surfaces

by

Michael Daniel Wu

## Thesis Readers

Reader

Glorianna Davenport, M.A.
Associate Professor of Media Technology
ABC Career Development Professor of Media Technology
Program in Media Arts and Sciences

Reader

Pattie Maes, Ph. D.
Assistant Professor of Media Technology
Program in Media Arts and Sciences

Reader

Robert Rowe, Ph. D.
Associate Director of the Music Technology Program
New York University Music and Music Professions

## Acknowledgments

There are many who deserve thanks. In particular:

Eric Métois, for an endless source of advice, ideas, small projects, diversions, and friendship.

Eric Jordan, for late night coding sessions, take-out, and videos.

David Waxman, for working on the early prototype with me.

The other people in the area, including Alex, Eran, Damon, Josh, Neil, Tom, and Joe, for making this place so stimulating.

Suzanne McDermott, for feeding me literature.

My sister Jean, for phone therapy.

My readers Glorianna Davenport, Pattie Maes, and Robert Rowe for their comments.

Liz Phillips, for showing me what interactive systems where all about.

And above all, Tod Machover, for giving me the space to grow and providing the vision for where my work will lead to.

## Contents

# 1 Introduction

Tools are the most visible fruits of technology and we employ tools in many areas of our lives. Jean-Louis Gassée expresses our enchantment with tools by saying: "We humans are in love with our tools because they help us become more than we are, to overcome our limitations and extend the boundaries of what it is possible to do with our brains and bodies [Gassée90]. It is significant that he uses the word "love" to characterize our relationship with our tools. With the advent of the computer, perhaps our most sophisticated tool, we are entering an age where we will no longer simply utilize, employ, or apply technology, but will also interact with technology. Whether we embrace the possibilities or are chilled by the implications, we must recognize that we will increasingly interact with technology in deeper and more intimate ways.

## 1.1. Motivation

A colleague and I badly play Mozart piano duets in such a way that only we could possibly enjoy. Yet, the experience is immensely satisfying. It is a chance to share an activity that encourages individual expression within a whole that exceeds the separate parts. Listening to skilled musicians playing music is also satisfying. That we are avid consumers of musical tapes and discs bears this out. Although one may listen actively, the act of listening is a passive one. In contrast, by playing music together, albeit at an amateur level, we enrich our relationship to music and the role of music in our lives. Even at this level, participating in a creative or artistic endeavor is a fulfilling experience.

Although hardly proficient pianists, my colleague and I have had enough musical training to make it through the duets without growing unduly frustrated. People who have not invested the time and money to acquire musical skills are confined to the role of passive consumers of music. A participatory interactive system blazes one path toward the inclusion of more people as active and expressive participants within a creative or artistic endeavor.

In its most glorious form, a participatory interactive system can be part of a kind of sacred space where people gather together and in which their actions have heightened significance. A responsive space where the interplay of actions between people and the space

dynamically shapes an experience that expands communal consciousness and perception while touching each participant in personal and individual ways. Such lofty aspirations are vital when one considers Brenda Laurel's lament:

> At the height of the Anasazi civilization in the great kivas, humans enacted spirits and gods. Long before these magical presences emerged from the shadows, dancing would begin on huge foot-drums whose throbbing could be heard a hundred miles away across the desert. The experience was an altered state that culminated in the performance in living presence. The great kivas are silent today. Even in our magnificent cathedrals, we hear only echoes of a magnitude of experience that has faded from our lives. There are no magical meeting places at the center of our culture, no sacred circles inside of which all that one does has heightened significance. In those few places where such transformations can still occur, the shadow of our civilization is fast obliterating the possibility. [Laurel93]

## 1.2. Interactive Systems

Interactive systems are characterized by the interplay of actions and responses unfolding in real-time between people and technology. For the person, there is an increased sense of immediacy because actions have heightened significance. Designers of interactive systems must create a relationship between a person engaging the system and the system itself. Each party must perceive, to a certain extent, what the other is doing. To sense what people are doing, interactive systems typically employ sensors and video cameras. Theoretically, any of the human senses could be involved, but interactive systems primarily rely on visual, audio, and tactile stimuli. A computer usually lies at the heart of an interactive system and executes a program that defines the behaviors of the system.

In his book Interactive Music Systems, Robert Rowe suggests a classification system for interactive music systems based on three dimensions that lie on a continuous spectrum [Rowe93]. In an effort to gain a deeper sense of interactive systems from a designer's perspective, I adopt Rowe's classification approach to the more general interactive system. It should be emphasized that the parameters for each dimension lie on a continuum of possibilities and the parameters used to describe each dimension generally fall on some extreme. The five dimensions an interactive system designer must address are the following:

- Goal: compositional - performance - experiential
- Expertise: novice - expert
- Control: instrument - director - collaborator
- Form: linear - multi-threaded
- Participants: individual - group

The first dimension is very broad and attempts to make loose distinctions about the encounter with the system or the goal of the designer in creating the system. It makes a distinction between compositional, performance, and experiential systems.

- Compositional systems engage the participant in a creative or design activity. Drum-Boy [Matsumoto93], an interactive percussion system allowing non-experts to build drum patterns with high-level tools, is an example. It should be noted that the design activity need not be so explicitly stated; it could be subtly woven into the entire experience of discovery and increased awareness of design materials, for instance.
- In performance systems, the person's role is that of a performer and implies a high degree of proficiency over the system acquired by rehearsal or advanced skill in a particular domain. Tod Machover's hyperinstruments, which extend the instrumental resources of virtuosi using sensors on traditional instruments, computers, and synthesizers, are an example [Machover92].
- Experiential systems focus on creating a particular experience for the person. The experience could be artistically inspired, entertaining, or educational. Experiential systems are about action and doing and what it feels like to participate within the interactive possibilities of the system. One example is Morgenroth's *Thinkie*, which is a form of interactive narrative that attempts to convey the experience of what it is like to think in a certain way [Morgenroth93].

The level of expertise required of a person to fully engage a system draws the distinction between novice and expert systems.

- Novice systems invite non-experts or people with no particular set of skills or knowledge to interact with the system.
- Expert systems demand a high level of skill or knowledge. One might have to spend a lot of time with the system to gain proficiency or have advanced knowledge in a particular domain, such as music. Video games usually fall within these two extremes.

A game may be simple to learn, but require practice to gain dexterity over the controls or to discover its secrets.

To qualify as an interactive system, the agents involved, both human and computational, must somehow act upon each other and change their behavior in response to the actions of others. The control dimension tries to determine the degree to which this is so and the nature of the mappings between the actions of the participant and the response of the system. We delineate this distinction with the instrument, director, and collaborator models.

- The instrument model invites the participant to play the system as if it were a musical instrument. In other words, control is at a low and detailed level and there is a clear and direct connection between action and response. A music system in this vein would offer precise control over individual notes.
- The director model offers control at a higher level than the instrument model. One example is the framework Houbart proposes for a "smart" VCR with content knobs [Houbart93]. To shape a video documentary on the Gulf War, the user would direct the flow of video material and the point of view using simple graphical controls. Behind the scenes, the system carries out the low level details of selecting the appropriate video material and ensuring that transitions are smooth.
- In the collaborator model, the system really flourishes as an interactive partner in its own right. Here, the actions of people do not necessarily provoke a direct response from the system, but may only influence the course or nature of subsequent interactions. The system sheds its subservient character, more appropriate to the other two models, to emerge as an interactive personality.

The temporal structure or form of the system or the order in which interactions unfold in time is the dimension that differentiates between linear and multi-threaded structures.

- A linear structure progresses through a system in a fixed order every time the system is run. In general, linear structures do not leave much room for interaction. However, there are systems that are structured linearly at a global level, but have interaction within individual sections. Machover's hyperinstrument pieces are divided into modes that generally correspond to sections of the piece [Machover92]. Each mode is executed in a fixed order, but a variety of interactions occur within each mode. Within Rowe's classification system, hyperinstruments would be score-driven.

- A multi-threaded structure provides each person with a unique path through the system. It is typically implemented as a branching strategy at various points in the system. *Limousine* [Weinbren85], Grahme Weinbren's interactive video system, adopts the metaphor of a drive in a limousine. The limousine will travel along routes established by the author, but at any time, the participant may intervene and direct the limousine to explore a different path. Collaborative systems, which include improvisatory ones, are multi-threaded.

Finally, the last dimension makes a distinction about the number of participants involved and provides a broad indicator of the social possibilities.

- Even though a system for an individual can only be used by one person at a time, there may still be a social context. Two people taking turns at a video game may discuss strategy and exchange tips as part of the whole process of interacting with the video game.
- A system for a group of people means that the designer must be aware of the social context surrounding the system. An important issue is how aware the participant should be to the actions of others and yet retain a sense of personal contribution.

## 1.3. Participatory Interactive Systems

Participatory interactive systems directly engage a non-expert in a creative endeavor or game-like activity. They afford intentional, shaped experiences; a participant must consciously decide to engage the system with actions that have heightened significance, and the designer of the system offers a mediated experience that empowers the participant with opportunities to shape, contribute to, and personalize the experience. The social context is collaborative in that form and structure emerge because neither the designer or participant has total control. Following the classifying framework outlined above, the participatory interactive system is experiential, novice-oriented, multi-threaded, and leans toward the collaborator model. Generalizations about the number of participants, however, are not appropriate.

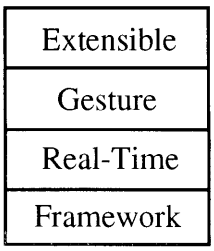
## 1.4. Responsive Sound Surfaces

As one possible model for a participatory interactive system, the responsive sound surface is a material surface, embedded with proximity sensors, that reacts with sound to human

movement. It creates a dynamic relationship between movement, space, and sound that engages a participant in a creative endeavor. To more fully define and explore the possibilities of this model, I have experimented with a responsive sound surface consisting of a bare, flat surface that has spatial sensing technology, based upon recent work by Professor Neil Gershenfeld [Gershenfeld94], hidden beneath its surface. The placement of four sensors formed a pattern of sound areas on the surface. The surface physically frames the overall sound texture and the sound areas are resonances within the texture that are associated with similar timbres or layers of sound. Participants engage the system with simple hand gestures above the surface in order to become increasingly more sensitive to the sounds, timbral changes, and layers within the sound textures and how their movements shape and influence these elements. A bare, flat surface may be disorienting, but by minimizing the visual element, the aural and kinesthetic ones are emphasized. By inviting people to touch air and feel space aurally, a special environment for reaching people with sounds and textures is created.

Toward that end, I have created a software environment that embodies an interactive approach to designing responsive sound surfaces and a set of examples built within that environment. Three principal elements define the approach to the software environment: dynamic languages, distributed system architecture, and software components. In this document, I discuss how a particular dynamic language, namely Smalltalk, becomes viable by using a distributed architecture along with carefully coordinated software components.

The software components provide the main functionality of the software environment and are conceptually organized as layers stacked on top of each other with higher layers building upon lower ones. At each level, I have identified, designed, and implemented initial sets of software components and protocols necessary to realize interactive experiences within the context of responsive sound surfaces. The figure below shows how they are organized.




Starting from the bottom, the layers with their functionality are as follows:

- Framework Layer: This layer enables the designer to hierarchically organize components by name; archive intrinsic, extrinsic, and symbolic component references; dynamically connect components; compose simple graphical interfaces quickly; implement interactions on the fly; and structure unfolding interactions.
- Real-Time Layer: Components in this layer provide scheduling, MIDI input and output, and timing. In addition, a suite of messages ensures the coordination of computers in a network.
- Gesture Layer: Included here are components that track direct and immediate gestures and others that follow gestures that build up and decay over time.
- Extensible Layer: To explore how extensible the software environment is, I have adopted and extended a set of compositional agents and principles from another software environment called Cypher [Rowe91].

Five example responsive sound surfaces are presented to demonstrate the effectiveness of the software environment and to illustrate the main concepts of this work.

After a survey of some of the interactive work relevant to the concept of responsive sound surfaces in section 2, there is a full discussion of the software environment and examples in sections 3 and 4 respectively. An evaluation of those two parts follows in section 5 and an outline of future directions in section 6 concludes this document.

# 2 Background

There are many artists, designers, and researchers exploring participatory interactive systems. As a result, the range of participatory interactive systems is broad. Video games have long headed their development and have spiraled into a multi-billion dollar industry. Science museums have embraced interactive multimedia technology and perhaps the most technically sophisticated systems that are accessible to the public appear in the award-winning permanent exhibition *Imaging, The Tools Of Science* at the Museum of Science and Industry [Reynolds93] [Bunish94]. Joseph Chung and Jeet Singh, concept designers of the exhibit, consciously avoid the traditional methods of presentation. There are no point and click interfaces and linear narratives where people mostly listen and follow along passively. Instead, they sought to engage people in a dynamic and immediate process that encouraged exploration. For instance, Face Net acquires the facial images of participants and sends them through a network to appear on various screens throughout the exhibit. Under control of the user, MetamorFace distorts, mutates, morphs, and blows up facial images to give a sense of the range of digital manipulations of images.

The work relevant to responsive sound surfaces focuses on systems that are experiential, novice-oriented, collaborative, and multi-threaded. In particular, they incorporate non-contact, spatial sensing or emphasize the role of sound. Here, I briefly survey some of the work pertinent to responsive sound surfaces. The survey is not comprehensive by any measure. Rather, it is meant to touch upon some of the pioneering work that influenced the conception of responsive sound surfaces and to outline a couple of the more intimate directions people are taking with interactive technology.

## 2.1. Myron Krueger

Myron Krueger's influential VIDEOPLACE [Krueger91] pioneered the use of video technology to superimpose the silhouette of a participant on a screen with computer graphics. One of the more popular interactions in VIDEOPLACE was CRITTER, an amusing artificial creature that cavorted with the participant. CRITTER's actions included floating down onto an open hand, dangling from an outstretched finger, and performing a celebratory dance on the participant's head.

In his book Artificial Reality II [Krueger91], Krueger proposes a new art form where response is the medium. In contrast to the passivity of audiences within traditional art forms, Krueger embraces an active role for the participant and stresses the interplay of gesture and response.

> It is the composition of the relationships between action and response that is important. The beauty of the visual and aural response is secondary. Response is the medium! [Krueger91]

Although often clever, Krueger's interactions are usually direct and simple. I intend to enrich the interactions with responsive sound surfaces by layering interactions at different levels. Simple layers would provide clear responses. More subtle interactions would only influence the system. For example, one might define the presence of a hand in a particular sound area as a simple switch indicating in or out and use presence to trigger events immediately when there is a transition between in and out. Another way would allow presence to slowly build up when a hand is inside the sound area or gradually fade away when the hand leaves. In this way, presence could gradually fade sounds in and out of a texture.

## 2.2. Pattie Maes et. al.

Recent work by Pattie Maes et. al has extended this kind of system to include 3D computer graphics worlds, artificial creatures with sophisticated behavior selection algorithms inspired by Ethological models of animal behavior, and hand and body gesture recognition across time and space [Maes et. al. 94]. Their Artificial Life Interactive Video Environment (ALIVE) system, immerses a human participant in a graphical world inhabited by autonomous behaving creatures that have their own goals and dispositions. Using natural gestures, the participant could act upon the creatures. For instance, a pointing gesture would send a Puppet away with a pout. What is fascinating about the ALIVE system is how it illustrates our increasingly more intimate relationship with technology. We are approaching a time where people will socialize with technology.

## 2.3. David Rokeby

David Rokeby also employs a video system to track people within a space. However, he expresses the responses of his system through sound and music. In *Very Nervous*

*System*, a hand-build video processor analyzes the output of a video camera to obtain a gesture history.

> Gesture histories are comprised of the shifts in dynamics (roughly equivalent to movement 'intensity', a combination of velocity and size of the moving object, a sort of 'momentum' perhaps). From this gesture history, more qualitative information is derived, to balance the purely quantitative character of the initial analysis. Therefore, the gathered information ranges from gross amount of perceived movement to time-based notions of consistency, unexpectedness, and rhythm. [Rokeby91]

Within IntAct, a real-time modifiable pseudo-object-oriented programming language, Rokeby defines the "interactive behaviors" that map the physical gestures to control commands that play various synthesizers. Like Krueger, Rokeby also elevates the role of interaction and discourages the auditioning of his music for its own sake as being beside the point. One must interact directly with his systems to get a real sense of what Rokeby is trying to communicate. Although I agree that interaction should be paramount, I'm inclined to believe that the quality of the system's responses should be very high in addition to the total interactions.

## 2.4. Sharon Daniel

Sharon Daniel's *Strange Attraction: Non-Logical Phase-Lock over Space-Like Intervals* is an electromechanical video and sound sculpture for two participants and two attendants that creates a metaphorical experience based on the model of a strange attractor in chaos theory [Daniel94]. Significantly, she expands the system's perception of its participants to include involuntary or automatic responses in addition to the intentional ones. Electronic devices monitor the heartbeat, respiration, or skin resistance of each of the four people involved. The gathered information triggered banks of samples that continuously changed based on each participant's response. Not only are we approaching a time where people will socialize with technology as in the ALIVE system, but interactive systems will also have a deeper sense of its participants by monitoring their unconscious and automatic reactions.

## 2.5. Liz Phillips

Liz Phillips is one of the pioneers of interactive sound installations and has profoundly influenced the conception and direction of this work. Phillips builds open systems that

reverently incorporate natural, organic elements within an adaptive environment. In her interactive sound sculpture installation *Graphite Ground*, a wooden walkway, pink Arizona flagstones serving as stepping stones, and large shards of raw copper are formally organized on top of raw wool that blankets the floor to create a contemplative site reminiscent of a Japanese garden [Phillips88]. Capacitance fields that are sensitive to the presence of people radiate from the copper rocks. Movement in a particular location and time activates changes in the pitch, timbre, duration, volume, and rhythm of the unfolding soundscape.

My experience working with Liz impressed upon me the importance of layering interactions at difference levels of transparency. Simple and direct interactions quickly give a participant a sense of the system at a basic level and serve as a point of departure for deeper exploration. Complex interactions enrich the experience and encourage deeper participation. At the time I was working with her, she was making the transition from an analog system to a digital system for the mapping of human movement to sonic events. Her analog system included an extensive collection of modules that could be mixed and matched with patch cords. We had no such collection of components in the digital domain and resorted to ad hoc, project-based solutions. Part of the work of this thesis was to identify, design, and implement an initial set of components that are relevant to interactive sound systems.

In many ways, responsive sound surfaces are a distillation or condensation of the fertile work Phillips has done with her sound tables, which includes *Mer Sonic Illuminations* [Phillips91]. Although Phillips articulates the responses of her system primarily through complex sonic events, her work has a very strong and important visual component. In contrast, the responsive sound surface I have been working with present a bare, flat surface without visual embellishment. The unadorned surface serves as a physical frame of reference for gestures made in space and its visual simplicity directs attention to the physical gestures made within its space and how they relate to sounds and layers within a texture.

# 3 Responsive Sound Surface Development Environment

The guiding principle that directed the realization of this work, from the software level to the nature of a participant's interactions with the system, is that simple elements may be combined in simple ways to produce complexity and rich interactions. Clearly, simple elements, especially in isolation, are easier to conceive and develop than complex ones. However, by combining and layering simple elements, one does not necessarily sacrifice complexity and richness. At the software level, the system was built up of simple software components that analyzed sensor data and play synthesizers. Although the individual interactions between the participant and the system were straightforward, they were layered on top of each other. Some layers were simple and direct and had a clear connection between action and response. Others were more subtle, where actions only influence responses or cause perturbations within the system. The mixture of simple interactions with subtle ones is an important ingredient of participatory interactive systems that avoid instruction or coaching. Clear and immediate interactions give the participant something to grab on to or a place to anchor further explorations into the system. Subtle interactions reward exploration and enrich the experience.

## 3.1. An Interactive Approach

I have created a software environment that embodies an interactive approach to designing responsive sound surfaces. At any time, the designer may modify the environment, along with a custom system built on top of it, and the changes will take effect immediately. Three principal elements define the approach to the software environment:

- Dynamic Languages: A dynamic language like Smalltalk incrementally compiles code on the fly so that programs may be developed while they are running.
- Distributed System: A distributed system functionally divides the work load among a group of computer within a network allowing systems to scale up incrementally.
- Software Components: The component culture approach emphasizes the building of reusable objects with explicit protocols. The resulting components may be mixed and matched to produce custom solutions.

### 3.1.1. Dynamic Programming Languages

One of the great benefits of dynamic programming languages is the ability to modify a program while it is running. Although many dynamic languages dynamically compile code changes, the penalty for such flexibility is performance speed that is slower than traditional, statically compiled programs. In addition, many dynamic programming languages, including Smalltalk, Common Lisp, and CLOS, have automatic garbage collecting facilities. Programmers may consume memory freely without having to worry about disposing it when no longer needed. When the system runs out of memory, it automatically reclaims the chunks of memory no longer referenced by the program. This takes time and perceptible pauses can be catastrophic in real-time computer-music applications. As a result, the computer-music community has generally shied away from dynamic languages for real-time work.

#### Hyperlisp

One notable exception is Hyperlisp [Chung91], a real-time MIDI programming environment embedded in Macintosh Common Lisp. Joseph Chung created Hyperlisp at the MIT Media Lab to develop the hyperinstruments conceived by Tod Machover [Machover92]. The later stages of hyperinstrument development demand fast, iterative refinements, often during rehearsals; the ability to change the system while it is running is not a luxury, but rather a necessity. The primary drawback of Hyperlisp lies within the implementation of the environment it is embedded. When Macintosh Common Lisp runs out of memory it typically spends at least two or three seconds collecting garbage. During a performance this can be a show stopper. Hyperlisp averts disaster by managing its own memory and requires its users to do the same. Unfortunately, programmers are notoriously bad at managing memory and the task is compounded by an environment that expects transparent mechanisms to manage memory automatically.

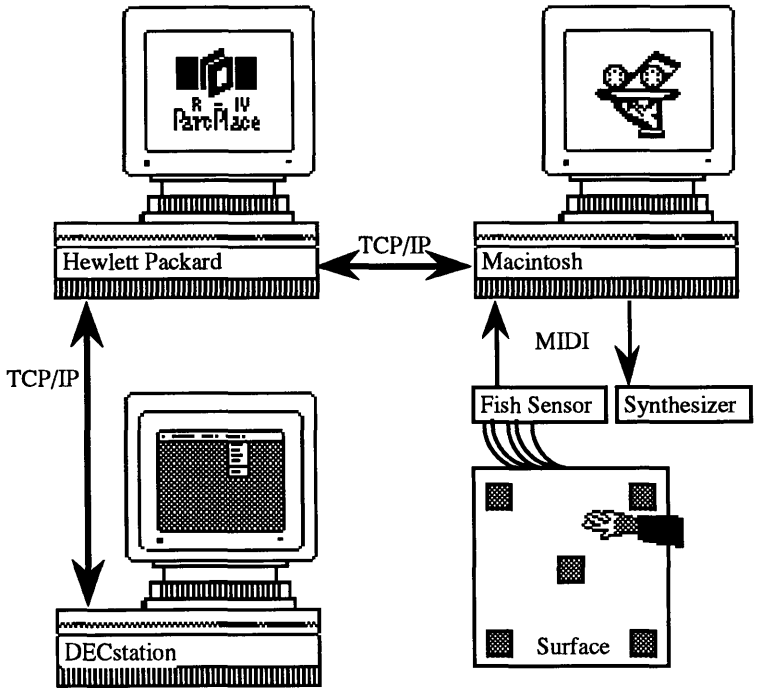
#### Smalltalk

Within the context of responsive sound surfaces, I discovered that by taking advantage of the exponentially increasing performance curve of hardware, it was possible to build real-time systems with a dynamic programming language and also take advantage of its garbage collecting facilities. I chose the venerable object-oriented language Smalltalk. I do not mean to suggest that Smalltalk is the only dynamic programming language suitable or that

the implementation I used, ParcPlace Objectworks\Smalltalk Release 4.1 [ParcPlace92a], is the only viable one. Rather, it is encouraging to find that it is possible to use a dynamic language for real-time work and to take advantage of all its features. I chose Objectworks\Smalltalk because of its mature implementation and its substantial class library. Hereafter, when I refer to Smalltalk in general, I will usually mean ParcPlace's implementation in particular.

### 3.1.2. Distributed Systems

A distributed system functionally divides the work load among a group of computers within a network. The RSS development environment is built upon a simple distributed system consisting of three computers: a Hewlett Packard 735 UNIX workstation running Smalltalk, a Macintosh IIfx with a C program I wrote that transferred MIDI data over the network, and a DECstation 5000 that was a display server. The computers communicated with each other using the industry standard TCP/IP protocol suite. The following diagram illustrates the various physical components of the RSS development environment and the communication paths.




RSS Development Environment

Although the distributed system is rather simple, it may appear to be unduly complicated for the work presented here. However, one of the goals of this undertaking is to demonstrate how to build an interactive system out of a network of computers and to start paving the path toward interactive systems that scale up to the size of Tod Machover's *Brain Opera* [Machover94b].

### The Midi Server

The MIDI server transfers MIDI data over the network. It is a C program I wrote that used a library called GUSI [Neeracher93] to provide a socket interface to TCP/IP. When Smalltalk connects to the MIDI server, the MIDI server sends clock ticks and sensor data, encoded as MIDI pitch bend messages, to Smalltalk every centisecond. This information is sent by way of unreliable, but fast, UDP packets. A lost packet is not catastrophic because an update follows in the next centisecond.

A distributed architecture helps to strengthen the areas where the dynamic language Smalltalk is weak. By having a MIDI server running on a Macintosh that transfers MIDI data over the network, I can guarantee with greater reliability that MIDI messages will be sent to a synthesizer at specific times. MIDI messages sent by Smalltalk are time stamped and sent over a reliable TCP connection to the MIDI server. Because the MIDI server has a scheduler, Smalltalk can stamp MIDI messages with times in the future and the MIDI server will send them to a synthesizer at the appropriate time. By running slightly ahead of the MIDI server (i.e., scheduling events in the future), Smalltalk can smooth out any latencies due to garbage collection. The delta time to schedule events in the future or scheduler advance is tunable and need not be the same for all the components within Smalltalk.

It is unlikely that any one language will be suitable for every situation. As much as I prefer to use a dynamic language, I also realize that a static language such as C is appropriate for many situations. The MIDI server illustrates this point; it is written in C for performance efficiency. It is also the only part of the system that is machine dependent because it uses MIDI routines that are specific to the Macintosh. MIDI libraries are not standardized across platforms and it is unlikely that they will be soon. One of the advantages of a distributed architecture is that I can isolate the machine specific MIDI server from the other parts. In addition to the HP 9000, Series 700 workstations, ParcPlace's implementation of Smalltalk runs on a number of other platforms including DECstations, Suns, and Macs. Therefore,

the more substantial Smalltalk piece of the RSS development environment may run on several platforms, some of which do not support MIDI, and I can be sure of MIDI access to synthesizers via TCP/IP, an industry standard. If it becomes necessary to implement the MIDI server on another platform, integration is straight-forward because of TCP/IP.

### 3.1.3. Software Components

In the design of the RSS development environment, I adopted a component culture approach where the emphasis is on building software components with object-oriented techniques. In his reflections on this relatively new culture of software development [Meyer92], Meyer contrasts the component culture with the more traditional project culture. The following abbreviated table from Meyer summarizes the relevant differences:

| | **Project Culture** | **Component Culture** |
|---|---|---|
| **Outcome** | Results | Tools, libraries, ... |
| **Goal** | Program | System |
| **Bricks** | Program elements | Software components |
| **Strategy** | Top-down | Bottom-up |
| **Method** | Functional | Object-Oriented |
| **Language** | C, Pascal, ... | Object-Oriented |

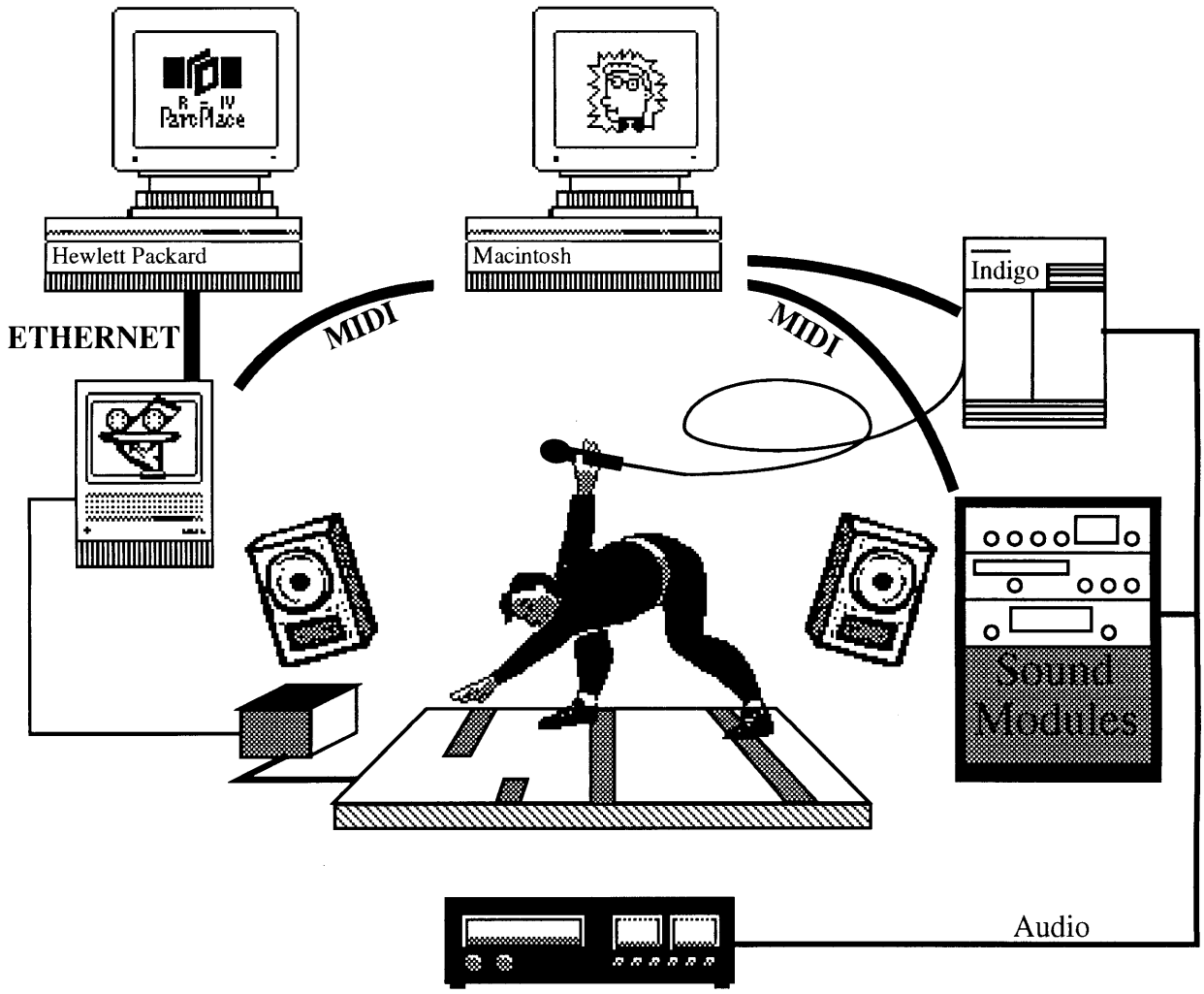
Within the project culture, the goal is a program that produces results that are just good enough to fulfill a top-down specification of requirements. Analysis proceeds from the specification to functional decomposition and data flow. Programs are made up of program elements, which are essentially modules built for the occasion. One of the classical languages is used for implementation.

On the other hand, the component culture makes an investment in software components to build tools and libraries for systems. The approach is bottom-up in that work begins at the component level and systems are built up from components. Using the fundamental object-oriented process of abstraction, objects within the application domain are abstracted into classes. In practice, a new component is initially too specialized and specific to a particular application and must be generalized upward to become reusable and extensible in different contexts.

Software components couple reusable, extensible computational entities with explicit protocols. The main body of this work was to identify, design, and implement an initial set of components relevant to responsive sound surfaces. This activity occurred at many different levels. Supporting components at a general level provide a framework for organizing cooperative software components. At the lower levels were components necessary for MIDI input and output and for the timing and scheduling of events. Included within the middle level were components for analyzing sensor data and generating sound textures. The highest level looked at formal approaches to structuring non-linear, unfolding interactive experiences and the components necessary to implement such strategies.

## 3.2. Prototype System

The figure below shows a prototype system David Waxman, Eric Métois, and I built to first demonstrate the concept of a responsive sound surface. One Macintosh running Max [Puckette & Zicarelli90] did all the high-level processing. The music and Max patches were written by Waxman. Waxman and I designed the interactions together. I wrote the C program on another Macintosh that transferred MIDI data over ethernet and the gesture analysis system in Smalltalk on a Hewlett Packard workstation. The predominant gesture was a beating action made with the same striking motion one uses to tap a table, except that physical contact with the surface is not necessary. Métois implemented a flexible, real-time sampling system on the Indigo and together, we built a proximity sensor box based on a design by Gershenfeld [Gershenfeld94]. Contrary to what the picture suggests, participants would only wave their hands over the surface. Our initial work had shown us that a responsive sound surface provides a rich context in which to design and experiment with interactions.

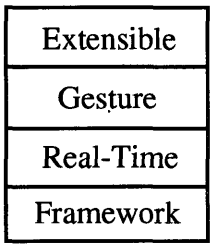



Prototype Responsive Sound Surface System (Courtesy of Eric Métois)

Waxman has since gone on to compose the music and interactions for a two person gesture instrument entitled the *Gesture Cube* [Waxman et. al. 94]. Proximity sensors were embedded within two faces of a large cube that was tilted on one of its corners. The corner was cut so the cube would be stationary. Through hand gestures and music, two participants engaged in an improvisatory dialog between themselves that was mediated by a computer running Max.

## 3.3. Software Component Layer Model

I have conceptually organized the various software components and protocols into four layers stacked on top of each other with higher layers building upon lower ones. The four layers are the framework layer, the real-time layer, the gesture layer, and the extensible layer. The figure below shows how they are organized.




The functionality of each layer is summarized below.

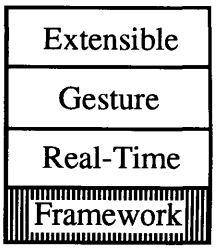
- Framework Layer: This layer enables the designer to hierarchically organize components by name; archive intrinsic, extrinsic, and symbolic component references; dynamically connect components; compose simple graphical interfaces quickly; implement interactions on the fly; and structure unfolding interactions.
- Real-Time Layer: Components in this layer provide scheduling, MIDI input and output, and timing. In addition, a suite of protocols ensures the coordination of computers in a network.
- Gesture Layer: Included here are components that track direct and immediate gestures and others that follow gestures that build up and decay over time.
- Extensible Layer: To explore how extensible the software environment is, I have adopted and extended a set of compositional agents and principles from another software environment called Cypher [Rowe91].

### 3.3.1. Accommodating Environment and System Designers

The software environment must accommodate two kinds of designers: the environment designer and the system designer. The first one seeks to extend the environment with a broader selection of components and resources to include in specific systems while the other mixes and matches reusable components to build custom interactive systems. Within the context of responsive sound surfaces, the environment designer works mainly at the extensible level This is not to say that the lower layers need no refinement or improvement,

but rather to stress that they are functional and one may build upon them in the role of either designer. The system designer treats the components in all the layers as fundamental building blocks that must be interconnected and cross-referenced in order for them to do anything useful. The design process for both is iterative and interactive because both the environment and a system built in the environment may be modified at any time.

## 3.4. Framework Layer




The framework layer coupled with the real-time layer provide the core functionality of a component-based interactive system. The real-time layer focuses on a set of services crucial for interactive system to run in real-time. The framework, on the other hand, directs its attention to the more mundane book-keeping issues of organizing components by name, archiving an entire system, connecting components together, and presenting a graphical user interface. Additionally, there are mechanisms for implementing interactions on the fly and structuring unfolding interactions.

The framework presented here is complete and comprehensive enough for a system designer to immediately build upon it within the context of responsive sound surfaces. In the grand tradition of Max, an iconic programming language named after computer-music pioneer Max V. Matthews, the names of most of the framework components begin with the prefix "Liz" in admiration of Liz Phillips' groundbreaking work on interactive sound installations.

### 3.4.1. A Component in the Abstract

At the heart of the framework lies the abstract class LizComponent. It has instance variables for a *name* and a *properties* dictionary, which is a data structure for associating keys with values. It contains the basic methods that all LizComponents must follow to work smoothly within the framework and is intended to be specialized or subclassed by the programmer. In general, when I refer to LizComponents, I mean instances of LizComponent and its subclasses. A suite of messages handles the life cycle of a LizComponent and consists of the following:

- setUp: The component has been created or the entire system has been reset. Do whatever initialization is necessary before the component can be started. When it receives this message, LizExternalClock will initialize a semaphore and start a process to service clients who are interested in knowing about clock ticks.
- cleanUp: The component is about to be released or the system is being reset. When it receives this message, LizExternalClock terminates its client process.
- start: The component should do its thing. This might be a one-shot activity, where a component might send configuration data to a synthesizer, or a continuous activity, where a component might continually reschedule itself to send out notes.
- stop: The component should stop doing its thing.
- tidyUp: CleanUp and stop often share some activities. For instance, components that play continuously after they have been started usually hold onto a task object returned by a scheduler. It is necessary to cancel that task during clean up and at stop time. Instead of writing the code twice, it is written once in tidyUp.
- referencesComponent: Return a Boolean indicating whether the receiving component references a particular component.

### 3.4.2. Hierarchically Organizing Components by Name

LizComponents are organized with a hierarchical component naming scheme and directory structure akin to the UNIX file system. LizDirectory holds a collection of LizComponents with unique names. LizDirectory is a kind of LizComponent so LizDirectory may be nested. The root directory is always named "/" and the slash character also serves as the pathname separator for components further down the directory tree. Apart from the obvious benefits of organizing components hierarchically, the naming scheme allows symbolic access to any component, which becomes important when archiving LizComponents.

LizSystem contains the root directory of all the LizComponents in the system. It has protocol for resetting and archiving the entire system; adding, removing, renaming, and moving components; and accessing any component by pathname. It is not a descendent of component LizComponent and therefore exists outside of the component directory tree structure. RssSystem extends LizSystem by adding an instance variable for *node*, which defines the configuration and interactions of a system.

### 3.4.3. Archiving Intrinsic, Extrinsic, and Symbolic References

After creating and configuring a set of components, the designer needs a way to store the system for later recall. The protocol for archiving LizComponents to a binary file simplifies the archiving of new component classes since each class need only concern itself with storage particular to itself and not worry about how its superclasses archive themselves.

When two components with references to each other are archived, we expect those references to be preserved when the components are unarchived. When the entire directory tree structure is archived, this is straight-forward in Smalltalk because I leverage classes that provide light-weight object persistence. ParcPlace's Binary Object Streaming Service (BOSS) can store a complex object, consisting of a root object and the set of objects reachable from the root, and restore the object with all references intact.

It would be convenient to use the archiving methods to implement cut, copy, and paste operations, but first a couple of issues must be addressed. If only a subset of a component directory tree is archived in a cut or copy operation, it is necessary to make a distinction between intrinsic and extrinsic component references. A LizDirectory holds a collection of references to other components. These references are intrinsic because they are inherent properties of the LizDirectory. On the other hand, a LizPortClient has a *clock* instance variable that references a clock component. A LizPortClient considers *clock* to be an extrinsic reference because it only provides a service and is not an inherent part of LizPortClient. When only a LizPortClient is cut or copied, the clock component is not archived with the LizPortClient. When it is pasted in, a LizPortClient would set its clock to nil.

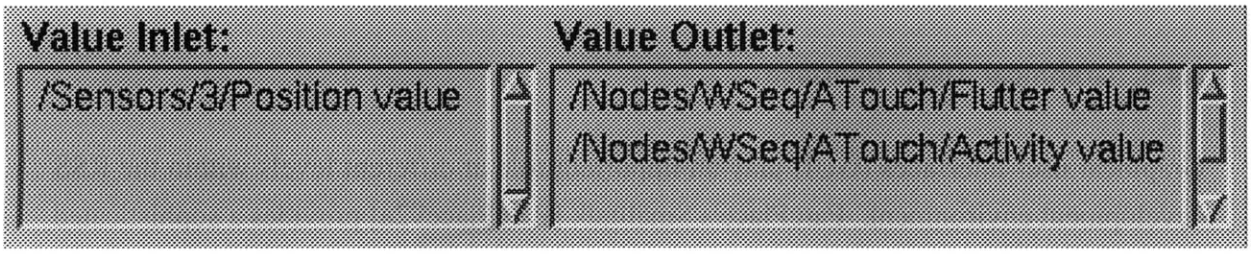
Other archiving mechanisms, such as the one found in NeXTSTEP [NeXT92], behave in this manner. I found the practice of setting extrinsic references to nil to be too restrictive and extended the approach by adding symbolic references. LizAspectPreset associates the aspects of components with values. When it is started, LizAspectPreset sets the aspect of its components to the corresponding values. The components are not an inherent property of LizAspectPreset because it makes little sense to copy out the entire components when only a LizAspectPreset is archived. However, copying a sole LizAspectPreset is useless because all the component references would be set to nil when it is pasted in. A symbolic component reference is the pathname the uniquely identifies a component. When a lone

LizAspectPreset is archived, its components are stored as symbolic references. At the time it is loaded back in, a LizAspectPreset would resolve each symbolic reference to an actual component or set it to nil if the object did not exist.

### 3.4.4. Dynamically Connecting Components

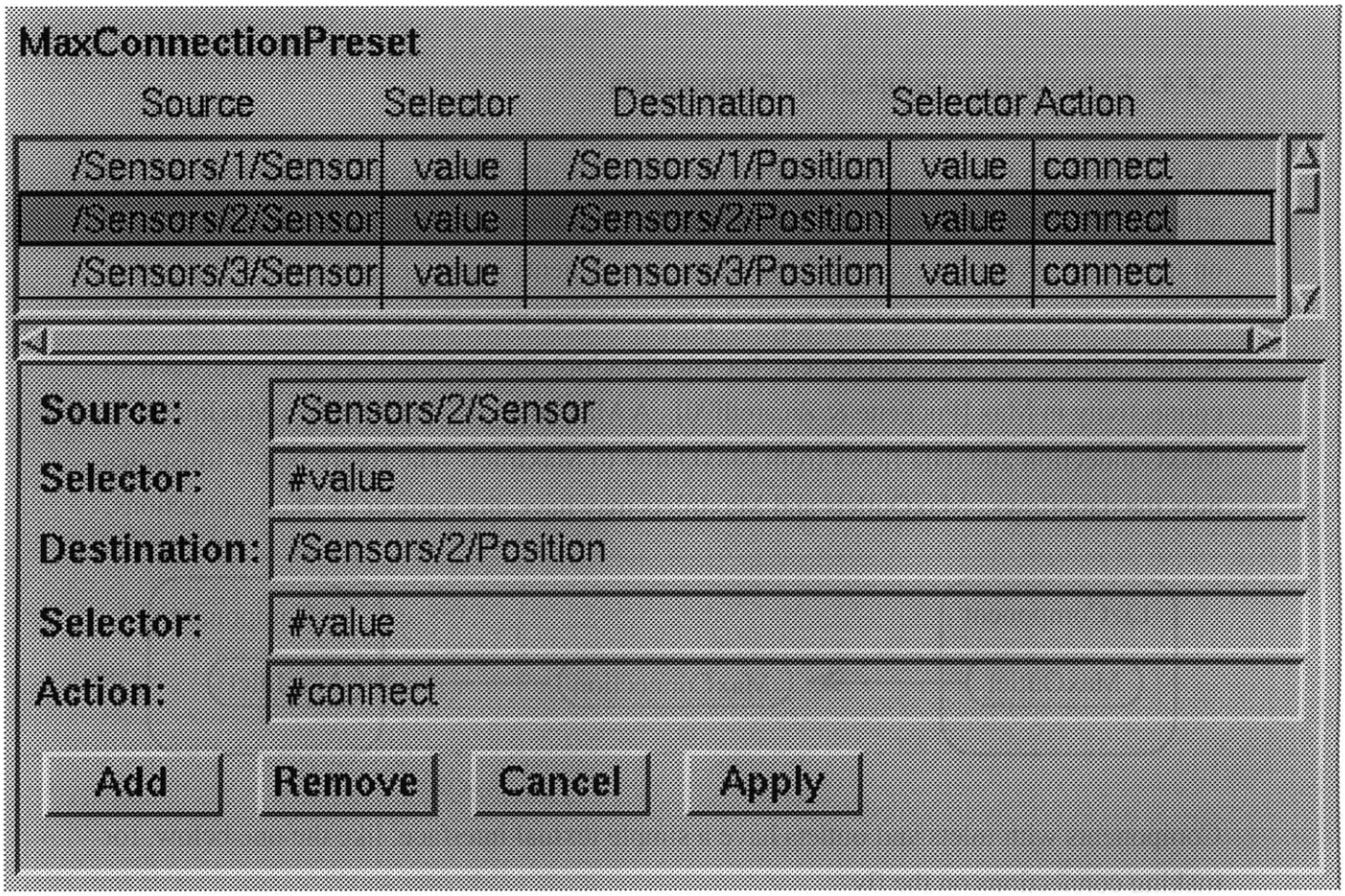
The primary approach for coordinating the activities of LizComponents is delegation and adherence to explicit protocols; a component has another component perform some actions on its behalf. When I began working on the components in the gesture layer, I found that in many cases, delegation with explicit protocols was unnecessary and simple pipelines would suffice. It was convenient for me to adopt the terminology and approach to connectivity in Max [Puckette & Zicarelli90]. A MaxOutlet is connected to a MaxInlet by a MaxConnection and data flows from MaxOutlet to MaxInlet. The figure below illustrates how the data flows. MaxOutlets and MaxInlets may have any number of MaxConnections.


Components with inlets and outlets have a simple textual interface, like the one below, for making connections on the fly.




The designer may establish or break connections at any time, either through the user interface or programmatically. When a system resets, all components discard their connections to revert to a clean state. Therefore, it is necessary to have a way of recalling connections between components and useful to reconfigure many connections in one step. MaxConnectionPreset maintains a collection of connections that are connected or

disconnected when it starts. Its interface is illustrated below. It is functional, but somewhat tedious to use.

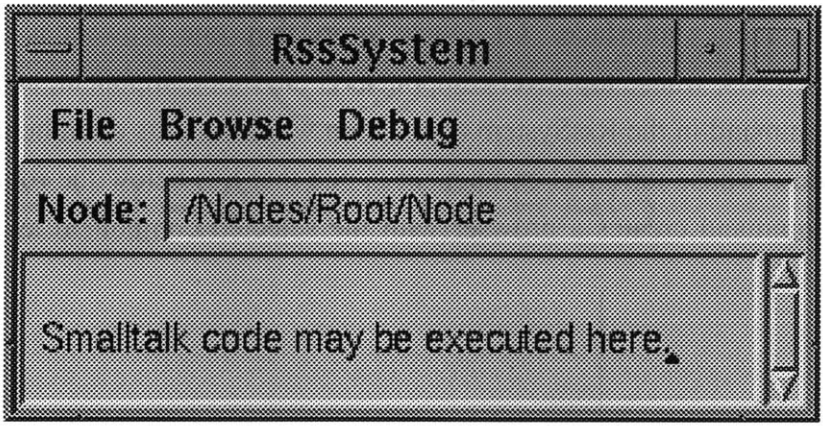



### 3.4.5. Graphical User Interface

Graphical user interfaces (GUI) play an important role at different levels within the RSS development environment. ParcPlace's Smalltalk [ParcPlace92a] provides a sophisticated set of integrated development tools for writing classes for components and other auxiliary objects. These include browsers, inspectors, interactive cross referencing, and a symbolic debugger. ParcPlace's VisualWorks [ParcPlace92b], built on top of Smalltalk, contains a graphical interface painter for lying out control components, such as buttons, sliders, and text fields, in a window. All these graphical tools were leveraged to build the RSS development environment. An added benefit is that all the graphical tools developed for the RSS development environment will work transparently on all the platforms supported by ParcPlace.

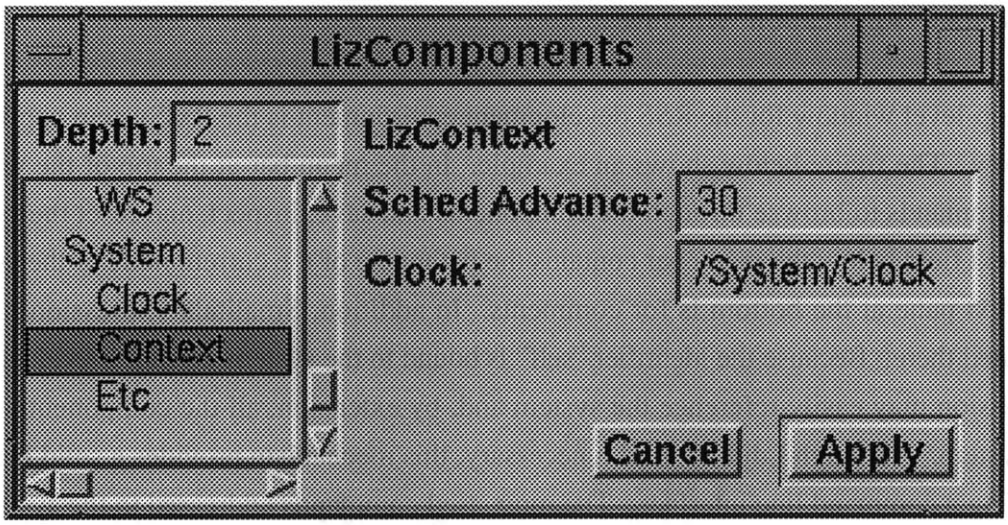
### RssSystem Application Window

The top-level window for the RSS development environment is the RssSystem application window. The figure below shows an example. In the menu bar at the top of the window are commands to browse through the components and save, inspect, and reset the system. The text field below the menu bar contains the pathname of the current node of the system. The text pane at the bottom is for comments. In addition, Smalltalk expression may be executed there. Because the identifier "self" is bound to the system when it is evaluated, an expression may easily reference the system and, by extension, all the components within the system. Although I did not find it necessary, this mechanism provides a solid basis for building a command line interface. The most used commands addressed to the system are readily accessible by menu. If I needed to manipulate a component directly, it was more convenient to use a LizPluggableComponent or an inspector on that it.



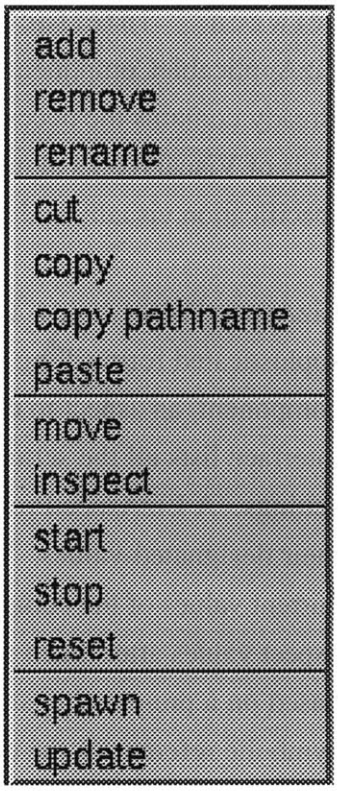

### Component Browsers

The browse command located in the menu bar of the RssSystem application window brings up a browser window like the one in the figure below. The list box on the left contains the names of components. The names are indented based on their level within the component directory tree relative to the top-level component of the browser. The number in the depth text field on the upper left determines the number of component levels to show in the list box. A value of 0 will list all the components below the top-level component of the browser.




The browser was designed to minimize window clutter. Instead of creating extra browser windows, clicking on a component name in the list box fills the right section of the window with that component's interface. Of course, more than one browser is often needed. Additional browsers, whose top-level component is the root component, are obtained via the browse command in the RssSystem application window. It is often helpful to have a browser window whose top-level component is one that lies below the root component. The spawn command creates a browser with the selected component as the top-level component.

The spawn command, along with other commands appropriate for selected components, is located in a menu that pops up when the middle button is depressed while the mouse pointer is in the list box. The figure on the right shows the choices of that menu. Instead of copying the component as an object, the copy pathname command allows the pathname of a component to be pasted into a text field at a later time. The move command prompts for a directory to move the component to and inspect brings up a standard Smalltalk inspector. Reset sends a **cleanUp** message to the selected component followed by a **setUp** message. If components are added or removed programmatically rather than through the graphical interface, browsers may become out of sync with the components of the system. The update command will make the browser consistent with the system.




#### Forms Filling Component Interface

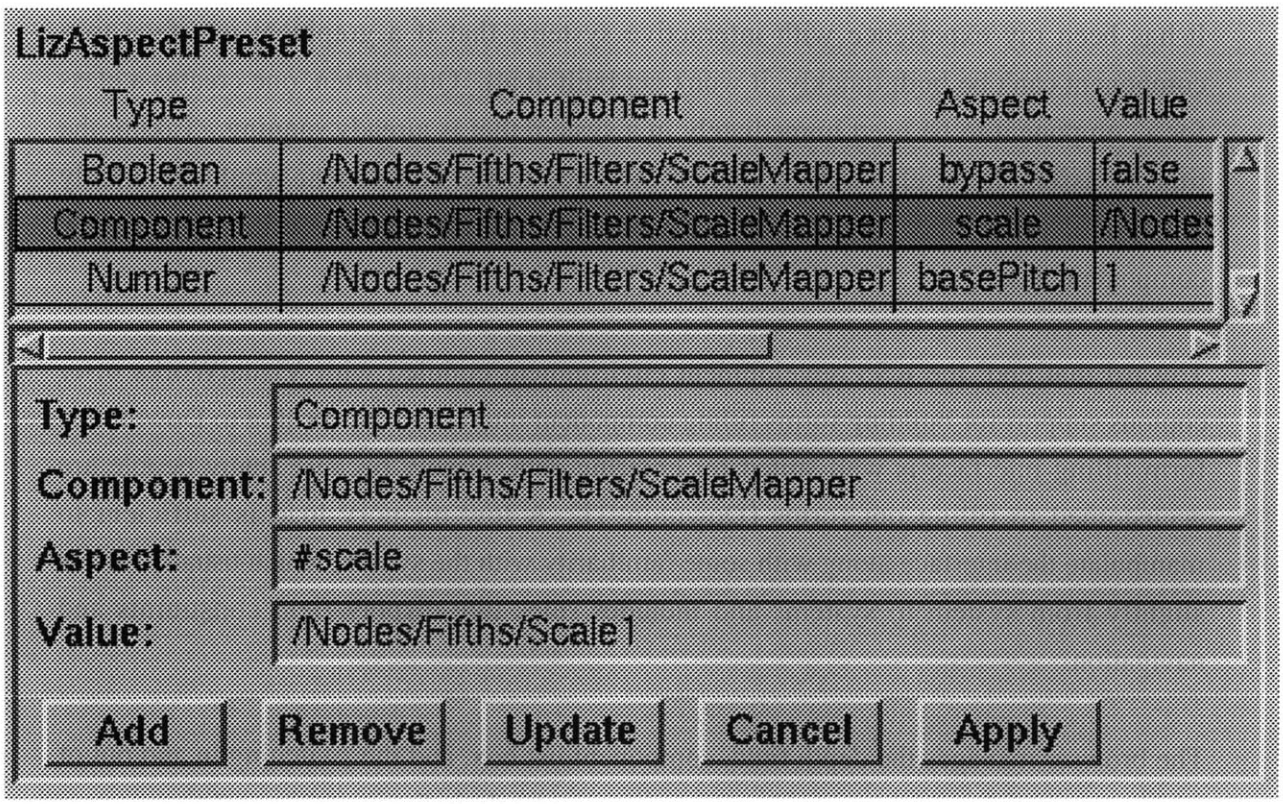
It has been my experience with interactive systems that if a graphical interface to a component can't be built quickly, it won't get built at all. Quite often, one creates a class for a component to be used immediately in a system that is already running. Taking the time to assemble an elaborate graphical interface to the component would be unduly disruptive, yet even a simple interface would greatly increase the utility of the new component class.

Toward that end, the RSS development environment coupled with VisualWorks offer copious support for building forms filling component interfaces. According to Downton, a form filling interface is well suited for entering data, largely because many people are accustomed to filling out forms [Downton91]. A low memory load is sufficient to manipulate the interface along with some familiarity with the syntax, which usually differs slightly from system to system and includes rules like display field constraints and navigation.

The browser manages the depth field, component list box, and the switching in and out of component interfaces. A new component class need only indicate which application class should govern its interface and supply an interface specification and a set of bindings

between interface elements and component parameters. LizApplicableComponentApplication is a suitable application class for a component and already contains the necessary logic for applying and canceling forms. The GUI building tools of VisualWorks along with its visual reuse capabilities enables one to create interface specification visually and swiftly.

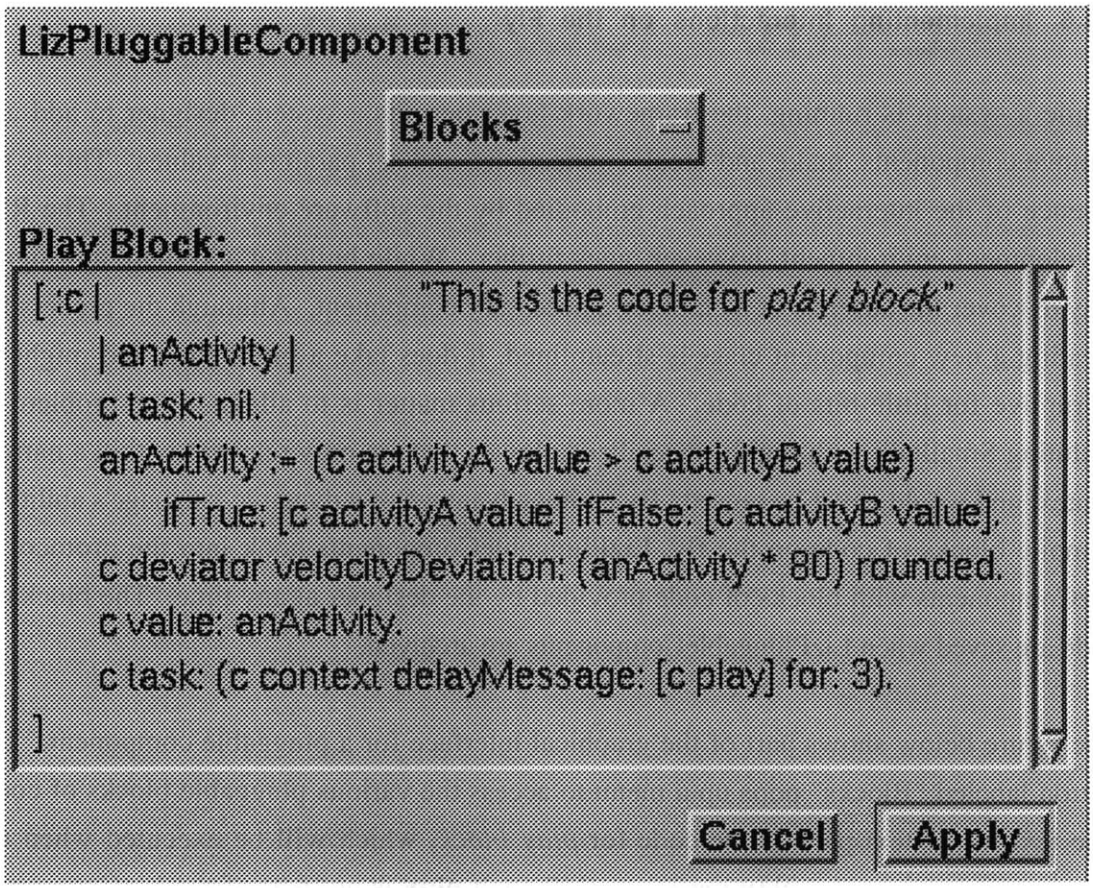
LizAspectPreset holds a collection of component settings and has one of the more sophisticated form filling interfaces, which is pictured below. A setting is composed of a component, an aspect to change, a value, and the type of the value. The upper half of the interface lists all the setting in a table while the lower half contains text fields for editing settings. A LizAspectPreset responds to **start** by setting the aspects of its components to their designated values. The **update** message causes LizAspectPreset to remember the current values of the component aspects in its settings. It is useful for taking a snapshot of aspect values after much parameter tweaking. An easy way to create additional snapshots based on a pre-existing snapshot is to copy and paste in the pre-existing one. Because of symbolic references, the components of the pasted in preset will be resolved properly. At that point, simply **update** it when appropriate.



### 3.4.6. Implementing Interactions on the Fly

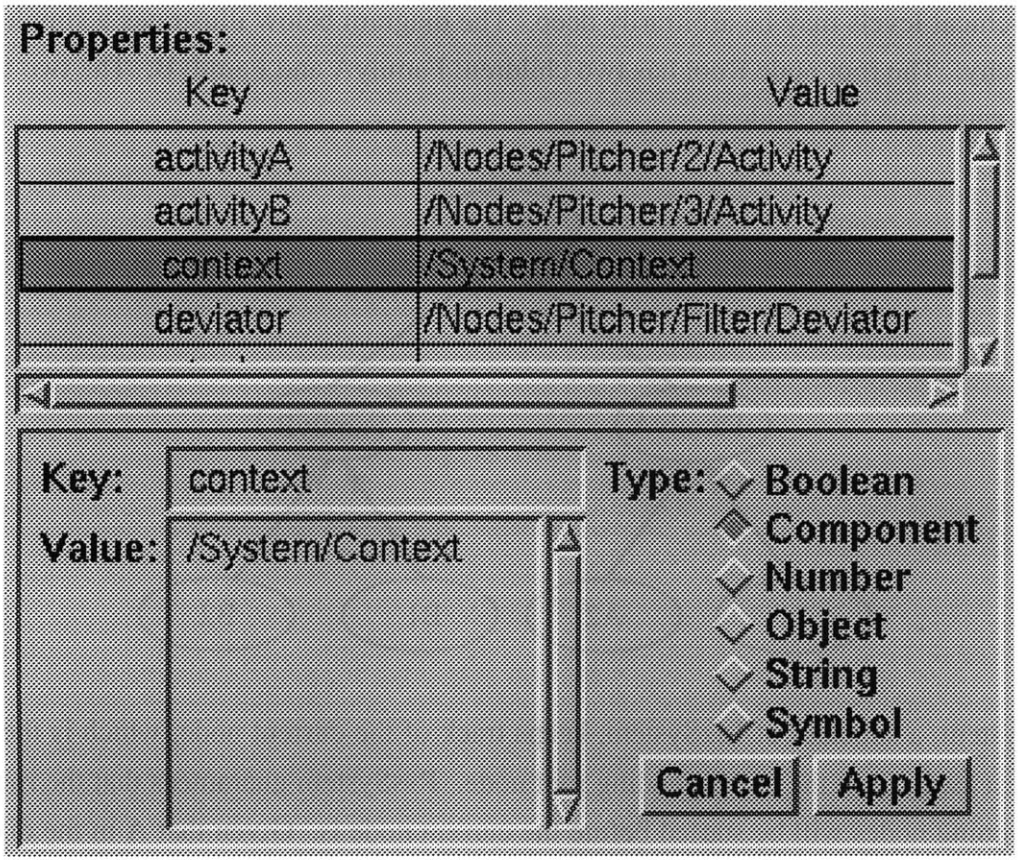
While a system is running, the framework layer permits a system designer to add and remove components, make and break connections, and set parameter values. This is the essence of the interactive approach embodied by the software environment. The RSS development environment, however, goes further. While building a system, a designer may discover the absence of a class for a necessary component. Although the process for creating a new component class is streamlined, it does take a bit of time and might be disruptive for little bits of code. A more serious matter is to alleviate the problem of polluting Smalltalk with highly specialized classes that are only relevant within a narrow context and would never be reused. By taking advantage of a dynamic language's ability to compile blocks of code on the fly, LizPluggableComponents hook into the protocol of LizComponents with blocks of code provided by the designer at any time. The blocks of code are saved when LizPluggableComponents are archived.

The figure below shows part of the graphical interface for a pluggable component. The button labeled "Blocks" pops up a menu to access the various pluggable blocks of code. The text in the text pane is the code for *play block*, which will be executed when the component receives the **play** message. Although **play** is not part of the LizComponent protocol, LizPluggableComponent includes a *play block* as a convenience for pluggable components that need to run continuously. One idiom for setting up periodic behavior is for the *start block* to perform any initialization and then tell the component to **play**. The *play block* would run through its calculations and then schedule itself to run again after some delay.




The *play block* of pluggable component.

For effective instance variables, the designer may employ the property dictionary that is part of every LizComponent. The graphical interface for accessing the properties of a LizPluggableComponent is shown in the figure below. The top half contains a table of all the keys and values in the properties dictionary while the bottom half has several widgets to set the value and its type for a key.




The *properties* of a pluggable component.

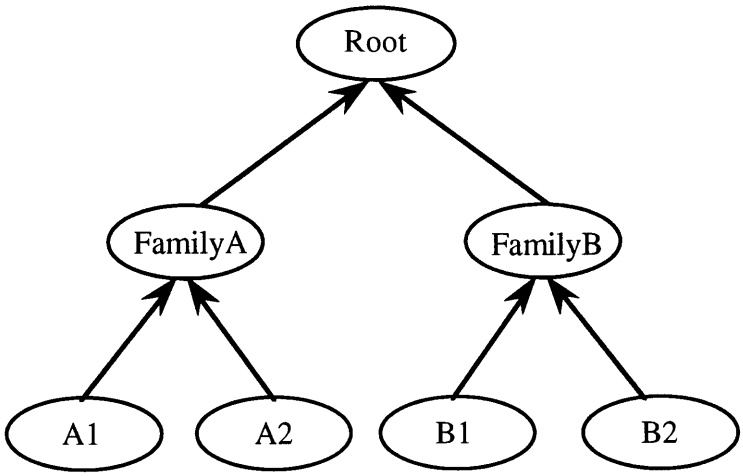
### 3.4.7. Structuring Unfolding Interactions

#### Nodes

LizNode provides a mechanism for structuring multi-threaded interactions. It is an extension of the mode in Machover's hyperinstruments [Machover92]. Modes define the configuration and interactions of a system from the time they are started until the time they are stopped. They generally correspond to a section of a piece and are executed in a fixed order. The interactions within a mode, however, customarily varied from mode to mode.

LizNode extends the idea of hyperinstrument modes by providing a hierarchical way of organizing nodes, similar to a directory tree. A LizNode may have a parent node that references another LizNode. A node without a parent is considered a root node and there may be more than one root node within a system, creating a forest of node trees. The

hierarchical levels may be arbitrarily deep. The diagram below illustrates a simple node hierarchy. The root node Root has two children: FamilyA and FamilyB. The two family nodes each have two children.




RssSystem, a subclass of LizSystem, keeps track of the current node, which is the node furthest away from the corresponding root node. RssSystem guarantees that the root and intervening nodes will be started before the current node is. Given that A in the diagram above is the current node, RssSystem would ensure that FamilyA and Root had been started before A1. To switch the current node, RssSystem locates the first common ancestor for the old and new current nodes. RssSystem then stops the old current node and its parents up to the common ancestor and starts the new current node and its parents up to the common ancestor. The common ancestor and its parents leading to the root node, which have already been started, are left untouched. Continuing with the example, to switch the current node to B1, RssSystem would identify Root as the first common ancestor. RssSytem would then stop nodes A1 and FamilyA and start FamilyB and B1.

One immediate benefit hierarchical organization offers is to help minimize the complexity of initialization and clean up at node start and stop time respectively. Root might set up the sensor and gesture component networks while FamilyA could start components shared by A1 and A2. When it becomes the current node, A1 may be confident that the sensor and gesture component networks and shared components have been set up and concern itself with initialization specific to itself. A switch to A2 would not disrupt the sensors and gesture component networks and shared components, allowing them to run continuously.

It is also invaluable, especially during development, to be able to jump to a particular state of the system in a clean and well-defined way. The RssSystem application window has a text field that contains the pathname of the current node. To move to another node, one may type its pathname within the field or choose a node from the node menu. We make a distinction between nodes that are abstract and those that are not. Abstract nodes are not intended to be the current node (i.e., the active node furthest away from the corresponding root node) of RssSystem and do not show up in the node menu. For instance, FamilyA might be an abstract node that only sets up components shared by its children.

Abstract nodes offer a structured way to handle interactive elements that must spill over the rigid boundaries of a node. In general, a node will start some components when it is started and stop them when it is stopped. If A1 and A2 behaved in this way, the transition from A1 to A2 would most likely be very discrete. To have elements bleed over, A1 might start some nodes and attach them to a key in the *properties* dictionary of FamilyA. A2 would expect to find those components in the *properties* dictionary of FamilyA and might manipulate their parameters while it is active and stop them when it is stopped. In other words, there is a structured way to share or pass components between nodes without resorting to global data.

The current node of RssSystem provides a mechanism for moving from one node to another. One of the tasks facing the designer is what policy to use to determine when to move to another node and which node to move to. Many early interactive video systems employed hardwired links between material. At branching points, dialog boxes popped up to ask the user which link to follow next. Clearly, periodic queries interrupt the flow of the interactions. In Grahme Weinbren's interactive video system *Limousine* [Weinbren], branching opportunities are deftly embedded within the video material with targets and the flow of the system is never disrupted. Targets define an invisible area of the screen. When a target is active and the participant enters the target, the system branches to the video material associated with the target. The system has an internal flow meaning that until the participant hits a target, the system will travel along routes established by the author.

The explicit links approach becomes unwieldy when the number of elements that must be linked together grows large. Adding a new element might require dozens and dozens of connections to and from that element, and reorganizing the network might mean tracing through a spaghetti of links. In his work on multivariant movies [Evans94], Ryan Evans favors a powerful description-based approach over hardwired links. To help movie makers

create movies that playout differently each time they are presented, he offers two tools: LogBoy and FilterGirl. LogBoy is used to create and edit descriptions which are attached to video clips in a database. FilterGirl is a tool for creating and editing filters that define descriptive playout constraints that guide the selection of video clips from the database. Filters take a set of video clips as input and return a subset of those clips as outputs. Filters become powerful when they are layered or combined in different ways, which include Boolean operators and temporal and contextual constraints.

### Node Sentries

Within the computer-music field, work to coordinate the actions of a computer and a human performer has followed two different paths: score-following [Vercoe84] [Dannenberg89] and score orientation [Rowe93] [Machover92]. Score-following strives for a tight synchronization between the computer and human performer by having the computer track the performer to a score stored in its memory. Score orientation loosely coordinates the actions of the computer and human performer by having the computer realign itself with the performer at certain landmarks or cue points within the composition.

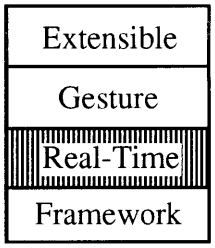
Although fixed, linear scores are inappropriate within the context of responsive sound surfaces, the concept of score orientation is useful in defining a policy for moving from node to node. LizNodeSentry keeps the spirit of score orientation by watching out for certain features in a participant's interaction with the system and advancing to its associated node when those features have been detected. Conceptually, one might think of LizNodeSentries as hawks that are allowed to hunt at certain times, either singly or in groups. When set free to hunt, a hawk quietly observes what is going on and will continue to do so until it is called back in or its prey surfaces, in which case it swoops down and moves the flow of interactions to another node. Hawks, however, are finicky creatures and have specific appetites regarding which prey they will pursue.

LizNodeSentry is a descendent of MaxPluggableComponent so any of the gesture components may be connected to it. Its instance variables include *wait time*, *duration*, and a *node* to move to when triggered. *Wait time* is the amount of time after it has been started to lie dormant before watching for the events it is interested in. When a node starts, it is often convenient to start a group of LizNodeSentries. Different wait times would stagger the activation times of the sentries. *Duration* defines how long a sentry is active from the time it begins its watch. If *duration* is nil, a sentry may be active indefinitely. In addition

to the pluggable blocks of code provided by MaxPluggableComponent, LizNodeSentry has *start watching*, *watch*, and *trigger* blocks. The *start watching* block is called when the sentry switches from waiting to watching. The *watch* block determines how the sentry observes what is going on. The *trigger* block is a hook for performing some actions before the transition to the sentry's node is made.

Recognizing the futility of trying to provide an exhaustive list of different sentries that would be applicable in all situations, the guiding principal behind LizNodeSentry was to provide an extensible framework for dynamically implementing specific node transition policies.

## 3.5. Real-Time Layer




The real-time layer offers a set of services that is crucial for interactive systems to run in real-time and coordinates the activities of the Smalltalk components with the MIDI server. These services include timing, MIDI input and output, and scheduling. The important point is that this functionality is already provided by the environment; the system designer can focus on crafting the interactions.

### 3.5.1. Services

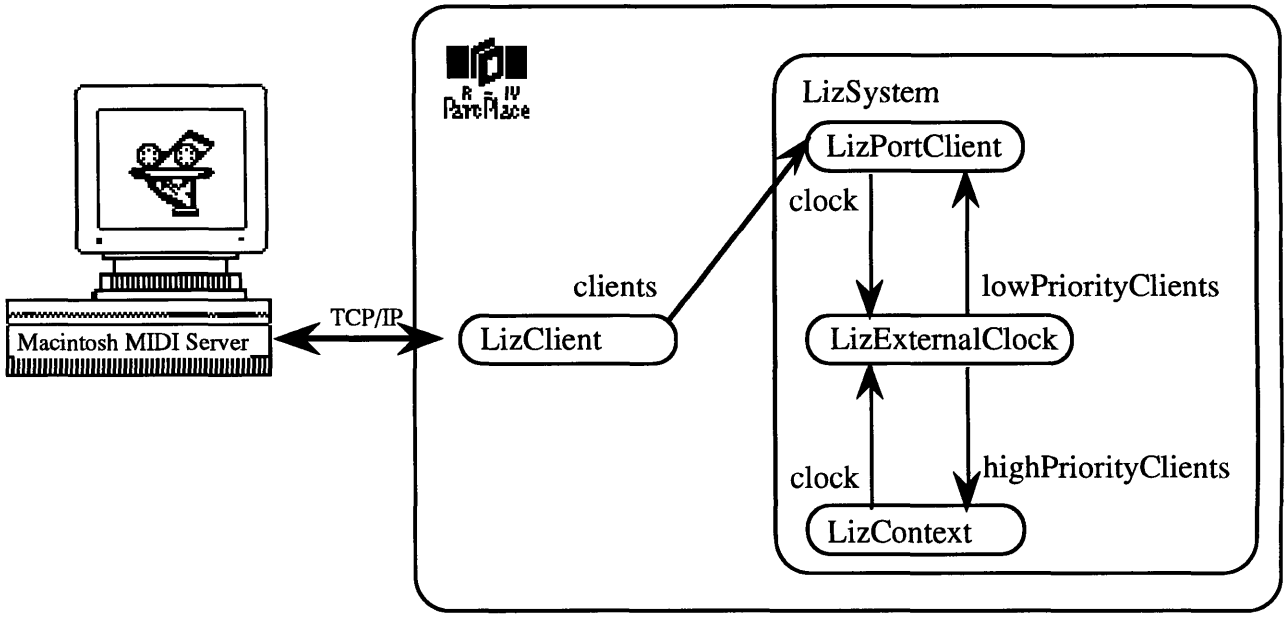
The components below supply the services of the real-time layer.

- LizExternalClock: It keeps track of the time by handling the MIDI clock messages from the MIDI server.
- LizPortClient: It provides MIDI input and output by transferring and receiving MIDI packets from the MIDI server via LizPort.
- LizContext: It provides a local context for scheduling tasks. Its scheduler advance determines how far ahead of real time the context will schedule events. Scheduling into the future helps alleviate latencies due to system load, but increases the system's reaction time. The ability to have multiple LizContexts increases the programmer's flexibility in meeting accuracy versus responsiveness requirements.

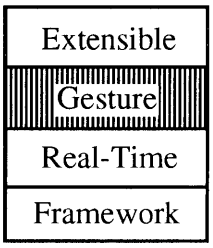
### 3.5.2. Coordinating With the MIDI Server

Enforcing the restriction that only one system is loaded into Smalltalk at any given time is optimal in terms of execution efficiency, but some compelling reasons for loading in two systems include the ability to cut, copy, and paste between the two and to try out versions back to back or simultaneously. However, it simplifies the programming of the MIDI server if we restrict the connections to one port number. To meet these demands, there is one LizPort object that connects to the MIDI server through one port number and may be shared by all the loaded systems.

A group of components works together to coordinate activities with the MIDI server to provide timing, MIDI input and output, and scheduling services. These include the LizPort, LizPortClient, LizExternalClock, and LizContext coordinate their activities with the MIDI server to provide the services of the real-time layer. Implementation details are located in the appendix. The diagram below illustrates the relevant relationships among the components.



## 3.6. Gesture Layer




The gesture layer contains all the components that analyze the sensor data for certain hand gestures. The approach to hand gestures in relationship to a surface was to think about simple things to look for that people could do easily and naturally without instruction or coaching. Each sensor location determines a sound area on the surface, and movements within a sound area provoke responses from the system. Conceptually, these sound areas are resonances within an overall sound texture that participants may influence and shape through their movements.

### 3.6.1. Direct and Immediate Gestures

Treating the sensor data as an evolution of measurements corresponding approximately to how close a participant's hand is to the surface, the obvious features to look for are position, velocity, and acceleration. Included at this simple level are a determination of whether a hand is within a sound area, the lowest or highest point of hand movement, and beats. Beats are made with the same striking motion one uses to tap a table, except that physical contact with the surface is not necessary.

RssSensor handles the sensor data from the MIDI server by sending through it outlet. To reduce jitter and minimize the effect of dropped sensor data packets, RssSmoother usually smoothes the raw sensor data from RssSensor and its output is treated as position values.

The organization of gesture components is an interconnected network where components pass data to each other via the Max-like connectivity objects. In order to determine beats, the beat component will need information from the velocity component, which, in turn requires data from the position component.

The remaining gesture components that look for obvious features are listed below:

- RssVelocity: Performs a linear regression of the last n values and multiplies it by a factor. The result is constrained to a value between -1 and 1 inclusive. N and factor are parameters. Generally, an RssSmoother that provides the hand's position is

connected to RssVelocity. Connecting RssVelocity to another RssVelocity results in acceleration.

- RssWithin: Output 1 if an input value is between minValue and maxValue, otherwise 0. MinValue and maxValue are parameters along with whether the boundaries are inclusive or exclusive.
- RssPeak: Tracks the highest input values.
- RssTrough: Tracks the lowest input values.
- RssBeat: Looks for beats which are determined by a tunable velocity threshold.

### 3.6.2. Gestures That Build Up and Decay Over Time

In order to add a more subtle layer to the interaction, there are components that look for presence, activity, stillness, and flutter. These features are characterized by their ability to build up and decay over time. The relevant components are described below:

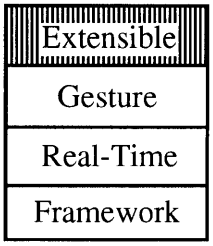
- RssPresence: While detecting whether a hand is inside or outside a sound area behaves like switch, presence will slowly build up when a hand is inside, or gradually fade away when the hand leaves.
- RssActivity: To get a sense of how much movement there is within a sound area, RssActivity averages distance over time.
- RssStillness: Stillness builds up when the position of the hand remains relatively fixed for a period of time. The medium value around which stillness builds up may be constant or may drift to the relatively stationary values.
- RssFlutter: Flutter builds up when the participant flutters the hand like butterfly wings.

### 3.6.3. Increasing the Effective Number of Gestures

As an initial set of gesture components, I feel it offers abundant possibilities, especially when one takes advantage of the ways to increase the effective number of gestures. For instance, the analysis does not necessarily have to span the entire height of a sound area, but could be restricted to a particular range of sensor values and thus form a resonant band within the sound area. RssSubRange only outputs the incoming values that fall within a specific subrange. So, RssSubRange could initiate a whole sub network of gesture components devoted to one part of a sensor's range. By correlating features from different sound areas, the complexity of the interactions grows significantly.

LizPluggableComponent and MaxPluggableValueObject are helpful in setting up very specific correlations that would be used only for a particular project.

## 3.7. Extensible Layer




The extensible layer marks the level where most extensions to the environment belong. Most new components and protocols that are intended for reuse in many custom systems fit here. Highly specialized components that are specific to one system generally take advantage of the pluggable components provided by the framework layer. To explore how extensible the RSS development environment is, I have adopted and extended a set of compositional agents and principles from another software environment called Cypher [Rowe91].

### 3.7.1. Cypher Components

Recognizing that a human participant naturally produces an unpredictable and dynamic environment for the sensors, we derive the complexity of the system's responses from the rich behaviors of human participants and keep the sound texture components relatively simple. Many of the sound texture components are derived from Rowe's compositional agents in Cypher [Rowe93] because they are particularly well suited for interactive manipulation, and also to demonstrate how easily some ideas from another body of work may be adopted and implemented within the RSS development environment.

Cypher is a compositional and performance interactive music system that behaves as a performer with its own musical personality and is therefore collaborative and multi-threaded. To use Cypher effectively requires a fair amount of musical knowledge. The two major components of Cypher are a listener, which analyzes one stream of MIDI music data, and a player, which articulates the musical responses of the system. These components are composed of a web of interconnected agents that are hierarchically layered. Agents have simple competences such as the ability to recognize the speed or density of the music it is listening to. When the agents are interconnected, the resulting whole exhibits remarkable behavior that belies the simplicity of its parts.

Rowe outlines three broad classes of machine composition methods that I summarize below:

- Sequencing: The playback of prestored musical material.
- Algorithmic Generation: Musical material is derived from seed material, which might be small collections of pitches or rhythmic elements, often using constrained random operations.
- Transformation: Musical material is systematically changed along some parameter.

Although it incorporates all three methods of composition, Cypher relies heavily on simple, straightforward modules that transform musical material and it is those modules that I cannibalize. Cypher accumulates changes in musical material by serially chaining transformation modules together; each module performs systematic operations on a block of musical events and then passes the block to the next module for further transformation. Complexity may be built up from simple elements when long chains are employed. All Cypher transformation modules uniformly accept three arguments: a *message* that indicates which one of two functions to apply, an *event block* consisting of an array of up to 32 events to be transformed, and an *argument* whose role depends on the message. The two acceptable messages are xform and mutate. The xform message indicates that the module should transform *argument* number events in *event block* and return the number of events in the block after doing so. The mutate message uses the value of *argument* to modify an internal parameter.

If the event block is restructured, a serial chain of transformation modules in Cypher maps naturally onto a pipeline of MaxValueObjects in the RSS development environment. Instead of separating the number of events in an event block and the actual events in the block and passing both pieces of information as two arguments, CypherEventBlock packages the *events* and the *size* of the events in one object. The *events* within a CypherEventBlock are implemented as an array of 32 CypherEvents where the *size* of a CypherEventBlock determines how many of the CypherEvents are relevant. CypherEvent mirrors most of the elements present in the event structure on which is it based on. It has a time *offset* relative to the previous event, a *chordsize* indicating the number of CypherNotes the event contains, and a *data* array holding 12 CypherNotes. CypherNote contains a *pitch*, *velocity*, and *duration.*

As a result of restructuring the event block, CypherFilter is a subclass of MaxValueObjects and the single MaxInlet and MaxOutlet provided by MaxValueObject is sufficient to create Cypher chains. CypherFilter has one parameter, *bypass*, which indicates whether the filter's transformation should be applied to event blocks received in its MaxInlet or pass the

event block through untouched. CypherPluggableFilter, a subclass of MaxPluggableValueObject, also has a *bypass*, in addition to the ability to accept blocks of code on the fly. If we adhered solely to Cypher's mutate protocol, we would limit ourselves to one modifiable parameter per module which, in many cases, would be too valuable to waste on *bypass*. By using other means, provided by the framework, to mutate parameters, we remove this restriction. LizAspectPreset would be convenient for grouping the settings of a group of filters in one component and restoring those setting in one shot. In addition, there are many ways to establish references to filters so that a component can manipulate a filter directly. The graphical interface to LizPluggableComponents provides a straightforward way to put a reference to a filter in its *properties* dictionary and to manipulate that filter with blocks of code submitted on the fly.

Following are brief descriptions of the filters adopted without or minor modification. More detailed descriptions may be found in [Rowe93]. Words in italic denote parameters accessible for modification.

• **CypherAccelerator.** CypherAccelerator shortens the durations between events by subtracting *downshift* from each offset in an event block except the first one.

• **CypherAccenter.** CypherAccenter accents every *strong* events in an event block.

• **CypherArpeggiator.** CypherArpeggiator breaks up chords by separating them into single note events temporally offset by *speed.*

• **CypherChorder.** CypherChorder builds a four note chord from every event in the event block.

• **CypherInverter.** CypherInverter modifies the pitches of the events in an event block by moving them about *mirror.*

• **CypherLooper.** CypherLooper repeats the events in an event block, taken as a whole, up to 2 + *limit* times. The offset between loops is determined by *speed.*

• **CypherPhraser.** CypherPhraser sub groups events in an event block by creating pauses every *length* events.

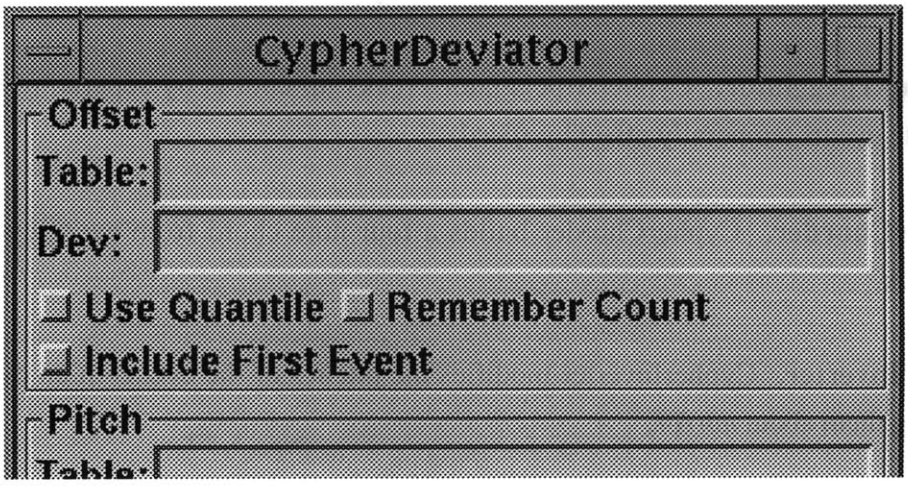
• **CypherSwinger.** CypherSwinger modifies every *swing* events by multiplying the offset time by *swing*.

• **CypherThinner.** CypherThinner reduces the density of events by removing every *thin* events.

• **CypherTransposer.** CypherTransposer offset the pitch of every note within an event block by *interval*. If *interval* is 0, the offset is a random number between 0 and *limit*.

I created a number of my own filters, which are described below. The bulk of the them take advantage of the ability to manipulate more than one filter parameter by applying a similar transformation, but with different parameters, to offset, pitch, velocity, and duration.

• **CypherDeviator.** CypherDeviator can independently deviate the offset, pitch, velocity, and duration of the events in an event block using separate deviation values that may be constant or from a MaxTable. The figure below shows the parameters that deviate the offset. Excluding the toggle for include first event, there are similar parameters for pitch, velocity, and duration.




For the moment, we will focus on how the offset parameters contribute in deviating the offset. CypherDeviator iterates through all the events in an event block to determine an offset deviation value for each event. If include first event is not set, the first event in the event block is excluded. To calculate an offset deviation value, CypherDeviator first checks for a component reference to a table. If there is a table, it is used to find the value,

otherwise the number entered for deviation is used. If a table exists and use quantile is true, CypherDeviator treats it as a probability table and the next quantile becomes the offset deviation value. When use quantile is false, CypherDeviator takes the next table value as the offset deviation value. CypherDeviator will cycle through table values across different event blocks, if remember count is set to true. Otherwise, it will always start at the beginning of the table for each event block.

If the offset deviation value is non-zero, CypherDeviator set the offset for the current event to a random number between the event's original offset plus or minus the offset deviation value. A similar procedure occurs for pitch, velocity, and duration using corresponding parameters.

• **CypherNoteDensity.** CypherNoteDensity reduces the density of events by keeping only a *percentage* of them.

• **CypherOffsetter.** CypherOffsetter offsets the offset, pitch, velocity, and duration of events in an event block using independent offsets.

• **CypherScaleMapper.** CypherScaleMapper maps the pitches of events onto a scale. For the *scale*, it expects a MaxTable of size 12, where each index corresponds to a semi-tone. An index with a non-zero value indicates that it is a valid pitch relative to *base pitch.* These two parameters are sufficient for CypherScaleMapper to map event pitches, that extend across the entire MIDI note range, while preserving the original octave. The mapped pitch is also constrained to *low pitch* and *high pitch.*

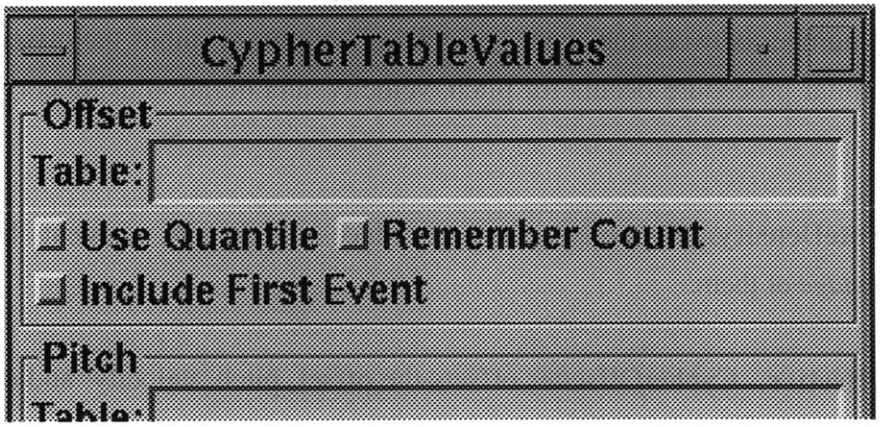
• **CypherScaleMapperWithOffset.** CypherScaleMapperWithOffset is provided in an effort to move away from purely chromatic pitch transpositions. It is a subclass of CypherScaleMapper and maps event pitches in the same way that CypherScaleMapper does. After mapping the original pitch, however, CypherScaleMapper adds *offset* to it by only counting pitches that belong to the scale.

• **CypherScaler.** CypherScaler multiplies the offset, pitch, velocity, and duration of events by independent factors.

• **CypherSynth.** CypherSynth converts CypherEvents to MIDI packets and sends them to a device that sends MIDI packets to the MIDI server. CypherSynths are generally at the

end of chains so that the final results are audible. CypherSynths at medial positions allow intermediate result to be heard.

• **CypherTableValues.** CypherTableValues uses independent MaxTables to determine offset, pitch, velocity, and duration values for events in an event block. The figure below shoes the parameters that apply to offset. Excluding the toggle for include first event, there are similar parameters for pitch, velocity, and duration.

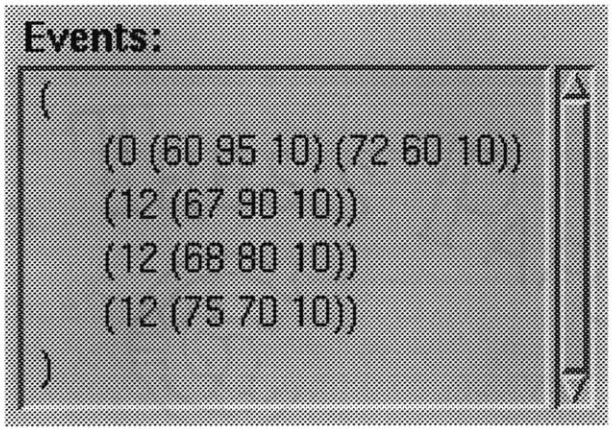



If there is an offset table, CypherTableValues iterates through all the events in an event block to determine a new offset for each event. If include first event is not set, the first event in the event block is excluded. When use quantile is set, CypherTableValues treats the table as a probability table and the next quantile becomes the offset. Otherwise, CypherTableValues takes the next table value as the offset. CypherTableValues will cycle through table values across different event blocks, if remember count is set to true. Otherwise, it will always start at the beginning of the table for each event block. A similar procedure occurs for pitch, velocity, and duration values using corresponding parameters.

Robert Rowe has used his Cypher program to analyze MIDI note data generated by a skilled instrumentalist and to transform that data using chains of transformation modules. The RSS development environment, on the other hand, expects sensor data corresponding to hand proximity. In other words, there are no MIDI notes, that come directly from the sensor data, for the Cypher components to transform. What follows are descriptions of components that initiate events into a Cypher chain.

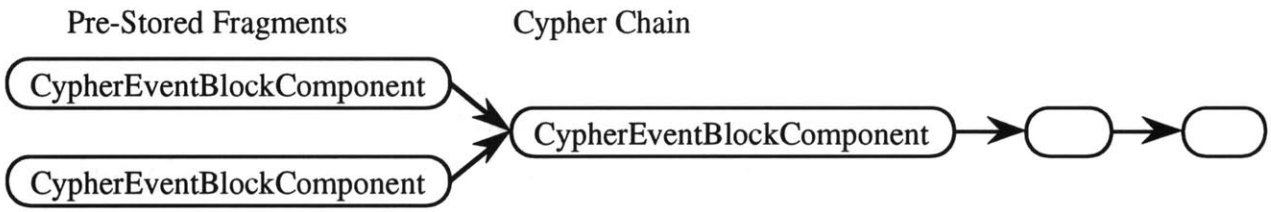
• **CypherEventBlockComponent.** CypherEventBlockComponent has an *event block.* The figure below shows the simple textual interface for specifying events in *event block.*

The entire list of events, individual events, and note specification are each enclosed in parentheses. In the figure, each line with numbers defines a complete event. The first number of each line indicates the offset time in centiseconds and up to twelve tuples, consisting of pitch, velocity, and duration, may follow. The interface is woefully crude and I consider it to be the bare minimum interface that is still usable. The point, though, is that I was able to build it very quickly out of classes provided by Smalltalk and move on to other things. If a more sophisticated interface is deemed necessary, someone can return later and add more graphical elements.

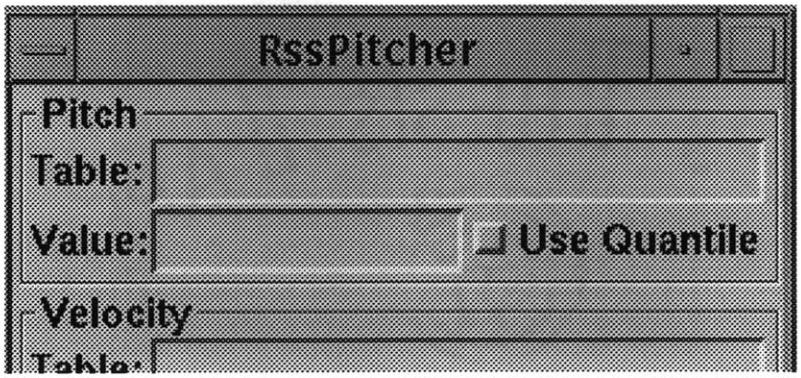



The intent behind CypherEventBlockComponent was to provide a way to send pre-determined fragments or seed material into a Cypher chain. When it is started, a CypherEventBlockComponent sends *event block* through its outlet. CypherFilters, however, generally destructively modify the event blocks they transform. After the first time of being sent through a chain, CypherEventBlockComponent's event block would probably be permanently altered and subsequent passes would further distort it beyond recognition. This is not necessarily unwanted behavior; it may be desirable to accumulate changes over time.

Nonetheless, there still must be a mechanism for preserving pre-stored material. To fulfill this need, CypherEventBlockComponent assimilates the event blocks it receives at its inlet before passing its event block through its outlet. CypherEventBlockComponent assimilates an event block by ensuring its event block matches it event for event. It does not modify the incoming event block or pass it through its outlet. The figure below illustrates how this would be set up. The CypherEventBlockComponent at the beginning of the Cypher chain acts as a buffer that may be destructively transformed.




• **CypherPitcher.** CypherPitcher generates events based on either constants or tables. The figure below shows the parameters that apply to pitch. There are similar parameters for velocity, duration, legato, and density.




To calculate the pitch for a note, CypherPitcher first checks for a pitch table. If there is a table, it is used to find the pitch, otherwise the number entered for value is used. Upon finding a table, CypherPitcher treats it as a probability table, if use quantile is set, and the next quantile becomes the pitch. When use quantile is false, CypherPitcher takes the next table value as the pitch. A similar procedure occurs for calculating velocity, duration, legato, and density using corresponding parameters. Minor differences include the multiplication of the duration by a user specified factor and the division of legato by 100.

When it is started, CypherPitcher sets up a recurring task that generates events. The period of the task is the duration of the generated event calculated at each invocation. The task generates one event whose chord size or number of notes is decided by the calculated density. The calculated duration sets the delay for the next invocation of the task. The calculated duration multiplied by the calculated legato, however, determines the effective duration for all the notes in the generated event. A new pitch is calculated for each note, but only one velocity value is computed for all of them.

• **CypherSequencer.** CypherSequencer cycles through its *collection* of components at a rate determined by *delta*. If *loop* is set, CypherSequencer cycles endlessly, otherwise it stops after the first iteration. When *send start* is set, each component will be started in turn. The *Output events* flag indicates whether each component should be passed through CypherSequencer's outlet. Although CypherSequencer was designed with CypherEventBlockComponents in mind, any LizComponent may belong to *collection* because all LizComponents respond to the message **start** and may be passed as values through inlets and outlets.

Following are descriptions of miscellaneous Cypher components.

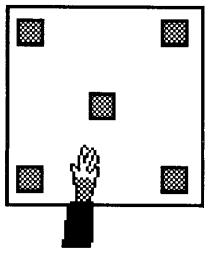
• **CypherDuplicator:** CypherDuplicator has two outlets with two corresponding event blocks. Incoming event blocks are assimilated by each of the two event blocks and the two event blocks are sent out the corresponding outlet. In general, Cypher chains tend to be serial with no branches. CypherDuplicator allows one to direct the results of one Cypher chain to two other independent chains. A pluggable version of this component permits one to submit a block of code that determines how incoming event blocks are split apart. It provides a solid framework for implementing a hocketing algorithm where the notes of a melody line are split up between two voices.

• **CypherEventBlockCopier:** A CypherEventBlockCopier has *source* and *destination*, which are both CypherEventBlockComponents. When it is started, CypherEventBlockCopier instructs the *destination* to assimilate the event block of *source* and the assimilated event block of *destination* is sent through its outlet. Incoming event blocks are assimilated by *destination*. The *output mode* determines whether the incoming event block or the event block of the *destination* is sent through the outlet. The intent behind CypherEventBlockCopier is to allow event blocks to be stored away and recalled at later points in time.

# 4 Responsive Sound Surface Examples

Several responsive sound surface examples were built from a single square foot sheet of Plexiglas and an early prototype of the quad hand sensor box, developed by Professor Neil Gershenfeld with Joe Paradiso, Tom Zimmerman, and Josh Smith [Gershenfeld94]. Because the sensors are based on the same physical principles fish use to sense their environment, sensor boxes are affectionately referred to as "Fish." The Fish sensor box has one transmitter and four receivers. It creates a weak electric field between the transmitter and each of the four receivers. The shape of the electric field between the transmitter and one receiver is very roughly approximated by that of a football. By disturbing the field with a living body and measuring the resulting fluctuations in the field, one may obtain an approximation of a hand's proximity to a surface. As a rough rule of thumb, the range of the sensor corresponds to the distance between the transmitter and receiver.

The geometry of the transmitter and receivers is very important in determining the sensitive areas of a surface. The layout used for all the examples described here is illustrated in the figure below. The transmitter was connected to the center electrode, made of a strip of copper tape underneath the surface. The receivers were connected to the electrodes in the corners. They were numbered starting from the upper left corner and moving counter-clockwise. For convenience, an electric fields is referred to as "sensor n", where n is the number of the corresponding receiver. Using two hands, this simple geometry allows one to independently disturb any two fields. By placing the forearm over one of the sensors in the front, a participant can produce a fairly steady value in that sensor while manipulating a sensor in the back with a hand. The center area around the transmitter is a "sweet spot" because a single hand may control all four sensors with small gestures.



Although all the examples described in the following sections are not necessarily connected, they belong to one RssSystem and share many components. Within the root directory, are the following subdirectories: System, Synths, Sensors, and Nodes. The components within System include the following:

- Clock: A LizExternalClock.
- Context: A LizContext.
- Port: A LizPortClient.

These three components have references to each other in order to synchronize properly with the MIDI server. They are also used extensively by the components in the examples to provide timing, scheduling, and MIDI I/O services.

The Synths directory holds one directory named WS. Within the WS directory are sixteen RssSynth components corresponding to the sixteen MIDI channels in the MIDI specification and numbered from 1 to 16. A Korg WavestationSR [Korg92] was the sole sound source. Its multi-timbral capabilities handle MIDI data on sixteen channels.

The Sensors directory contains four sub directories corresponding to the four sensors and labeled from 1 to 4. In addition, it has two monitor components that graphically display the raw sensor values and a simulator. The simulator has four sliders, corresponding to the four sensors, that can generate sensor values. It was an invaluable development aid when the Fish sensors were not available.

Inside of each of the four sensor sub directories are the following components:

- Sensor: An RssSensor that fielded MIDI sensor values from the sensor corresponding to its parent directory.
- Position: An RssSmoother that smoothed the sensor values from Sensor. To reduce jitter and minimize the effect of dropped sensor packets, components interested in proximity measurements should take them from Position instead of Sensor.
- Monitor: A LizNumericValuesMonitor showing Sensor and Position values.

The Nodes directory contains sub directories of all the examples. In addition, there is a sub directory titled Root and another one named Randomness. Included in the Root sub directory, a LizNode called Node is the parent of all the other nodes in the system and

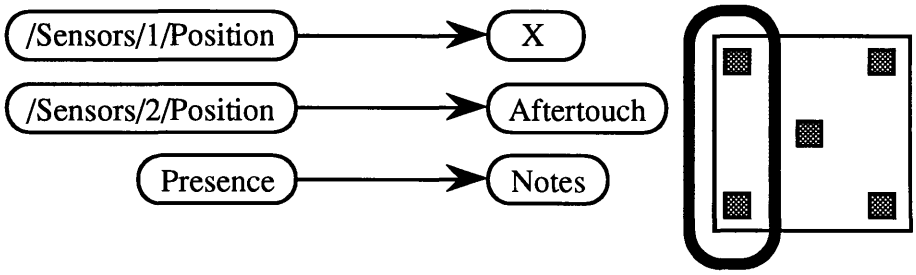
therefore, will be started before any other node. It starts the sensors and establishes basic sensor connections so that its child nodes may be confident that the sensors are already running when they are started.

The Randomness sub directory has a node that starts a component which generates random MIDI notes. It is used as a basic system test.

## 4.1. AmpMod

The AmpMod example takes its name from the technique used to modulate the sounds on the Wavestation, namely amplitude modulation. It is a straightforward demonstration of how to map the sensors to MIDI control messages and how the right sounds belie the simplicity of the mappings.

The surface is split down the middle into left and right halves. Although the two halves control separate sounds, their mappings are mirror images. Here, we describe the left half. The subdirectory named Left in the AmpMod directory contains the components that are pertinent to the left half, which include X, Aftertouch, Presence, and Notes. The connections are diagrammed below.




X is an RssControlChange component that maps the incoming position values from sensor one directly to MIDI controller 16 values. The sound on the synthesizer responds to controller 16 messages by mixing between two timbres, one of which is tied to the left speaker and the other to the right. The incoming position values from sensor two are directly mapped to aftertouch values by Aftertouch, an RssAftertouch component. Aftertouch MIDI messages control the opening and closing of a filter on the synthesizer.

Presence is a MaxPluggableValueObject that builds up presence if a hand is within range of sensor one or two. RssPresence, a component class that exists already, is inadequate here

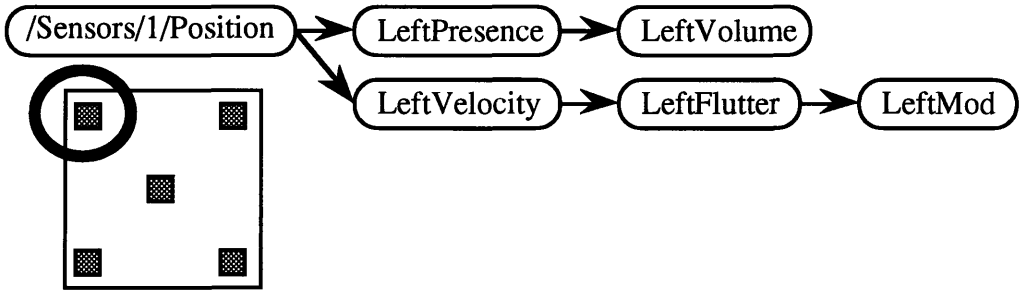
because it only looks for presence in one sensor. Instead of creating a subclass of RssPresence for somewhat special purpose components that looked for presence in two sensors, a MaxPluggableValueObject is more appropriate; two blocks of code and several properties are sufficient. The *start block* simply ensures that it has been stopped, sets its *value* to 0, and tells itself to play. The *play block* builds up or decays *value* based on whether a hand is within sensor one or two and sends *value* through its outlet. It also instructs *context*, a property in its *properties* dictionary, to send it the **play** message one centisecond later.

Presence connects to Notes, another MaxPluggableValueObject. When its incoming values indicate that there is presence, Notes generates three notes by selecting three consecutive pitches from a pitch array starting at a randomly selected index. The time stamps of the notes are offset to achieve a rolling chord effect. Notes includes some logic to ensure that the generated notes last for a minimum duration before it creates new ones. Because the algorithm for generating notes and the mappings are simple, the effectiveness of this example relies heavily on the sounds.

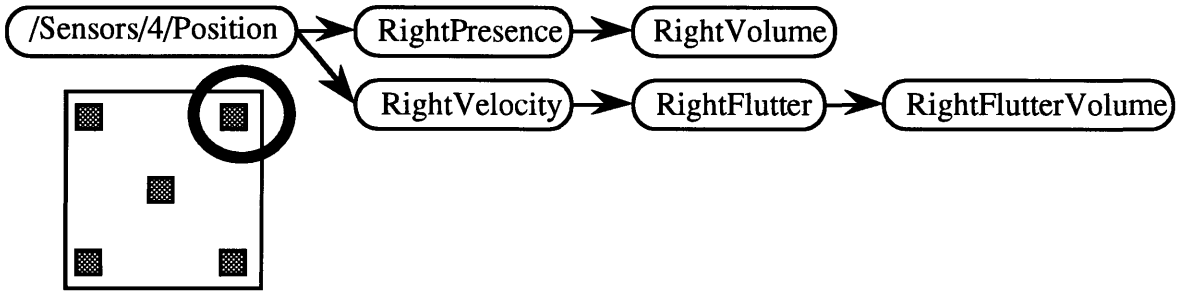
## 4.2. Ancient

In the example named Ancient, different interactions are assigned to the different sensors. Again, the surface is functionally split through the middle, except this time into upper and lower halves. The upper half controls various pedals tones and sounds and its components are located in the Pedal subdirectory of Ancient. When Ancient is started, a LizPluggableComponent named Notes, in the Pedal subdirectory, will send out pedal notes on several MIDI channels. These pedal notes will last until Ancient is stopped and are not immediately heard because their MIDI channel volumes are 0.

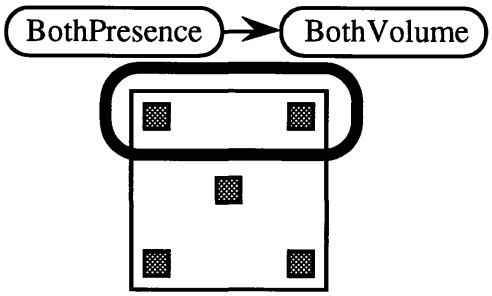
The top left sensor controls the pedal note on MIDI channel 1. Its connections are diagrammed below. LeftPresence feeds into LeftVolume to control the volume of the pedal tone and LeftFlutter connects to LeftMod to determine the pitch modulation depth.




The top right sensor controls the pedal notes on MIDI channel 2 and 7. Its connections are diagrammed below. RightVolume controls the volume of MIDI channel 2 which has a soft, repeating timbre separated by silence. The sound on MIDI channel 7 is a sequence of metallic and inharmonic timbres and RightFlutterVolume sets its volume.

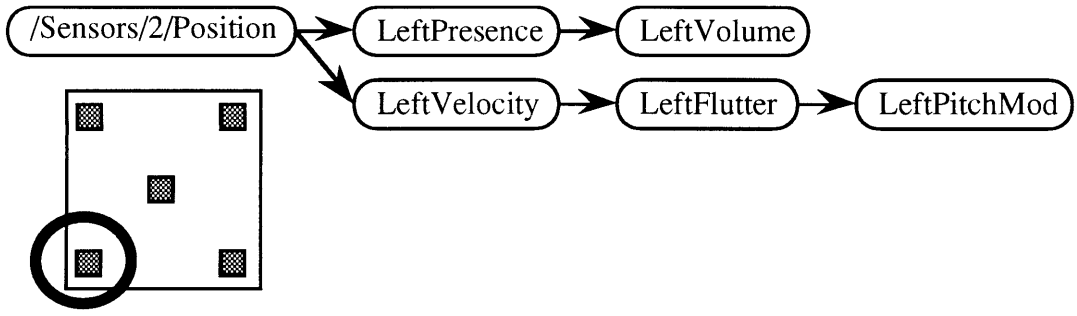



The top two sensors control the remaining pedal note on MIDI channel 3. The sound is an intermittent sequence of metallic timbres filled with long gaps of silence. BothPresence is a MaxPluggableValueObject that builds up presence if a hand is in either of the two sensors. It connects to BothVolume which determines the MIDI volume on channel 3. The diagram below illustrates what is happening schematically.

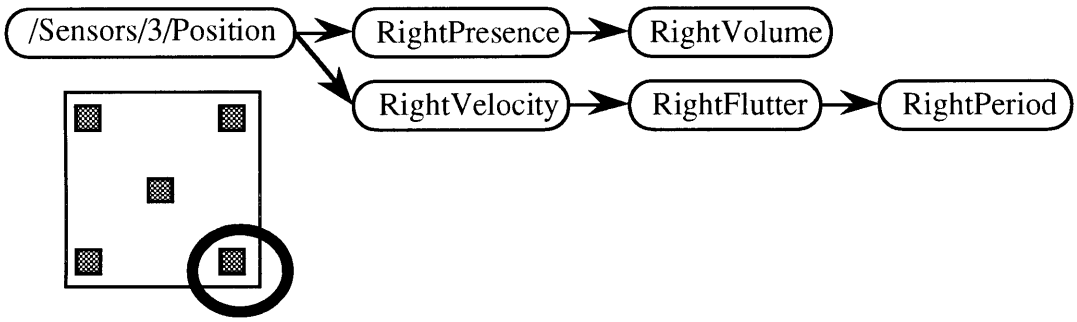



The lower half of the surface plays a synthesizer sound named AncentCelsia on a couple of MIDI channels. Its components are found in the Celsia subdirectory. When Ancient starts, a LizPluggableComponent named LeftNotes plays single notes from a table, treated as a

probability table, named PitchTable. LeftNotes generates a new note every ten seconds and holds each note until it plays another one. The bottom left sensor controls those notes and its connections are diagrammed below. Presence shapes the volume of the notes and flutter determines the pitch modulation depth.




A LizPluggableComponent named RightNotes also starts when Ancient does. It is identical to LeftNotes except that it plays notes on a different MIDI channel and its period for generating new notes, initially forty centiseconds, is much shorter. The bottom right sensor controls both the volume of the notes and the period for generating them. Its connections are diagrammed below. Again, presence shapes the volume. Flutter, however, connects to RightPeriod, a MaxPluggableValueObject. RightPeriod scales the period of the generated notes by the incoming flutter values; enthusiastic fluttering spins out a flurry of notes.

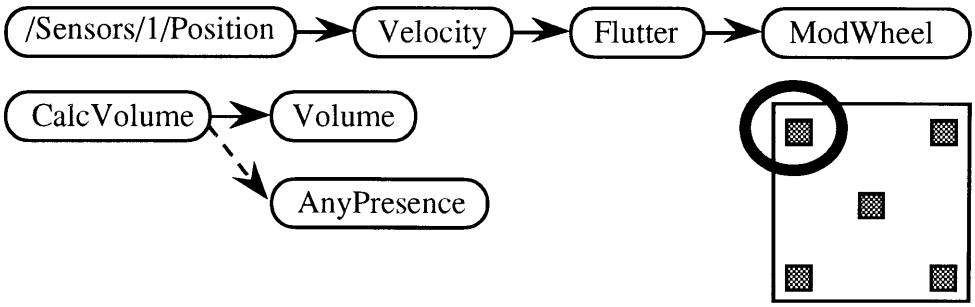



## 4.3. WSeq

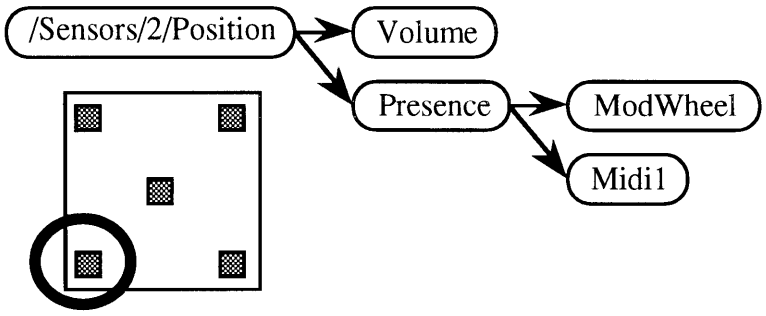
As with Ancient, the WSeq example assigns different interactions to different sensors. In addition, many interactions are layered onto sensor three. Each sensor is associated with its own sound. In ascending order of the sensors, the sounds are Galax2, Spectra, ATouch, and Spectrm respectively. There are subdirectories within WSeq with the same

names as the sounds. They contain the components relevant to the corresponding sensor/sound pair.

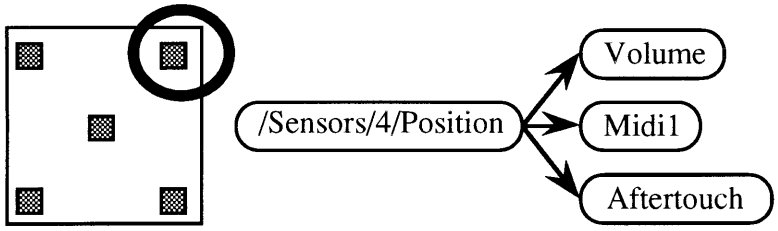
The top left sensor controls the Galax2 sound. Its connections are diagrammed below. When the example starts, two drone notes are held until the example ends. Flutter feeds into ModWheel to control the pitch modulation depth of the sound. CalcVolume is a MaxPluggableValueObject that connects to Volume. The dashed line from CalcVolume to AnyPresence indicates that CalcVolume references AnyPresence, but does not pass any data to it. AnyPresence, a MaxPluggableValueObject, builds up presence if a hand is within the range of any of the sensors. At every centisecond, CalcVolume uses the value of AnyPresence to scale the volume of Galax2 to one fifth of the MIDI volume range. However, if a hand is within sensor one, it shapes the volume directly. The idea is to have the drone notes build up to a low volume when a hand is present anywhere on the surface, but to also allow direct shaping of the amplitude when a hand is in a particular sensor.



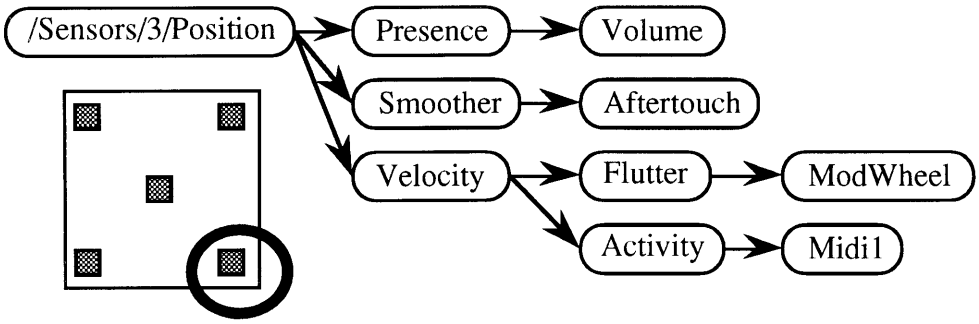

The bottom left sensor controls the sound named Spectra, which contains many inharmonics. When the example starts, Notes, a LizPluggableComponent, plays a note from a probability table every 500 centiseconds. Position shapes the volume and Presence builds up pitch modulation depth via ModWheel. Presence also builds up Midi1 values. Midi1 is a RssControlChange component that converts incoming values to MIDI controller 91 messages, a controller designated as MIDI 1 on the Wavestation synthesizer. Spectra responds to these controller messages by adjusting the depth of random filter modulation.



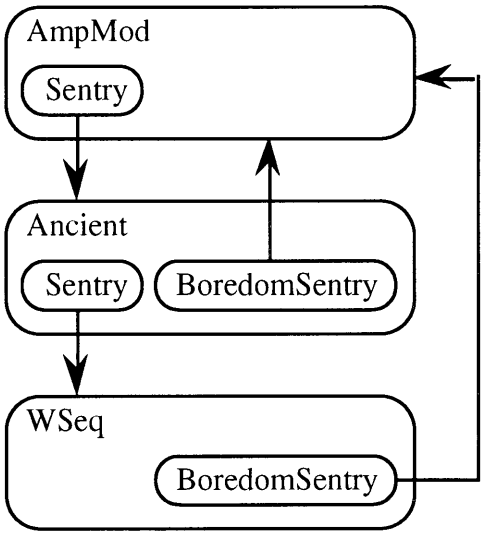

The top right sensor controls the sound named Spectrm and is similar in simplicity to the one just described. When the example starts, Notes, a LizPluggableComponent, plays a note from a probability table every 500 centiseconds. Position shapes Volume, Midi1, and Aftertouch. Again, Midi1 adjusts the depth of random filter modulation. Aftertouch selects which timbre in a sequence of waves is heard.



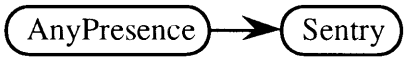

The bottom right sensor controls a sound named ATouch. The connections, diagrammed below, show that many interactions are layered onto the sensor. When the example starts, Notes, a LizPluggableComponent, plays a note from a probability table every 500 centiseconds. Presence builds up Volume and Smoother heavily smoothes out position values before Aftertouch maps them to MIDI aftertouch values. As with the previous sensor, aftertouch selects which timber in a sequence of waves is heard. The Smoother makes the transitions between timbres more gradual. Flutter, via ModWheel, determines the amount of pitch modulation and Activity feeds into Midi1 to control the depth of random filter modulation.



AmpMod, Ancient, and WSeq are linked together by several node sentries. The figure below outlines the various transitions. AmpMod has one sentry that triggers a transition to Ancient. Ancient has two sentries: Sentry and BoredomSentry. Sentry advances the system to WSeq and BoredomSentry returns to AmpMod. WSeq only has BoredomSentry which moves the system back to AmpMod.



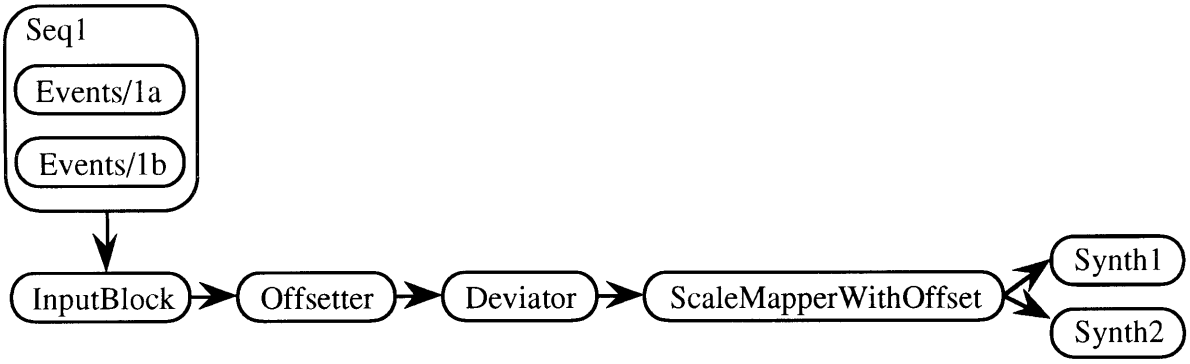

All the Sentry components behave in the same way. As the figure below reveals, the output of AnyPresence, a MaxPluggableValueObject that builds up presence if a hand is within the range of any of the sensors, is connected to Sentry. After waiting for ten seconds, Sentry watches the presence values to see if they stay above a threshold for duration of six seconds at which point it will trigger a transition to its node. Over time the duration of six seconds is scaled down so that the longer a node is activated, the faster Sentry will trigger in response to presence values above a certain threshold.



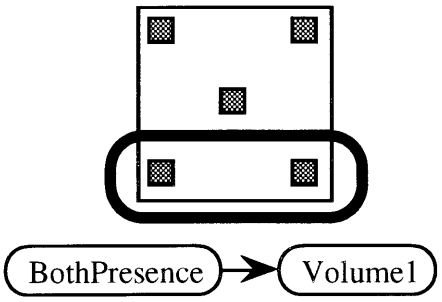

In contrast, the BoredomSentry's are much simpler. As the figure below shows, AnyPresence also connects to BoredomSentry. Whenever there is no presence for fifteen seconds, BoredomSentry triggers a transition to its node.

## 4.4. Fifths

The Fifths example takes its name from the predominant interval of the seed note material. It employs a chain of Cypher components, shown in the figure below. Seq1 is a CypherSequencer that has a collection of two CypherEventBlockComponents: 1a and 1b. When the example starts, Seq1 continuously cycles through its collection starting a component every forty centiseconds. InputBlock, a CypherEventBlockComponent, assimilates incoming event blocks into its own event block and passes its own event block down the Cypher chain to be destructively modified and played on two MIDI channels. This arrangement preserves the seed material in 1a and 1b.

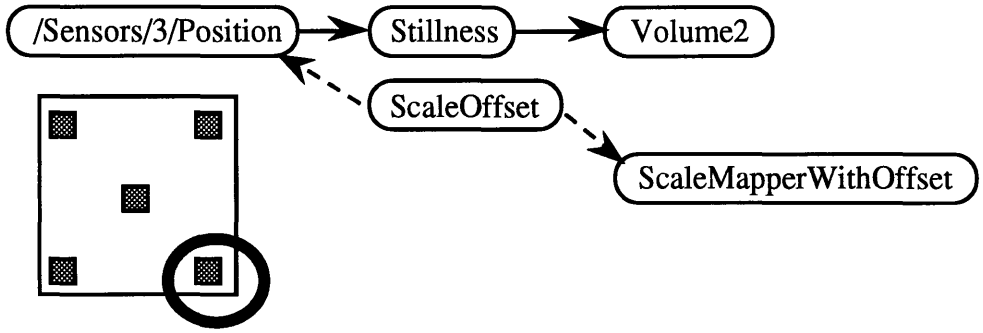



Only the two bottom sensors are used. As the figure below shows, BothPresence, a MaxPluggableValueObject that builds up presence if a hand is in either of the two bottom sensors, feeds into Volume1, an RssControlChange component that controls volume on the MIDI channel corresponding to Synth1.

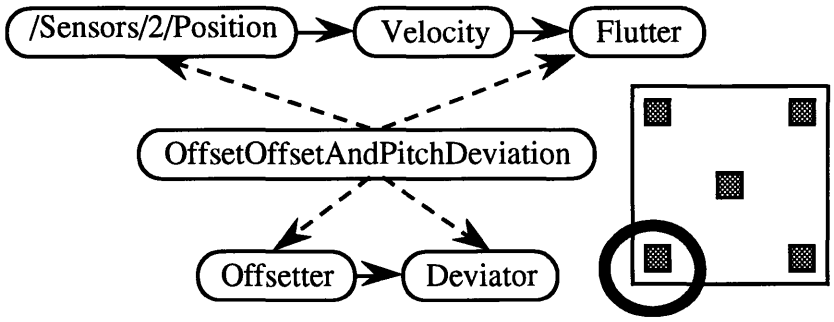



As the figure below reveals, Volume2, which sets the volume of the MIDI channel corresponding to Synth2, is determined by the Stillness on the third sensor. Every eighty

centiseconds, ScaleOffset, a LizPluggableComponent, maps the position of sensor three to the offset of ScaleMapperWithOffset to transpose the events blocks by scale intervals.

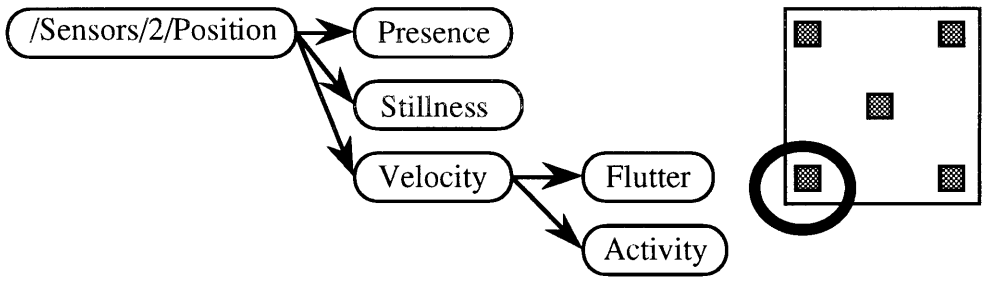



Finally, OffsetOffsetAndPitchDeviation, a LizPluggableComponent, references the position and flutter of sensor two to calculate an offset offset for Offsetter and pitch deviation for Deviator. If there is a lot of flutter, OffsetOffsetAndPitchDeviation sets the offset to 18 centiseconds and the pitch deviation to span two octaves. Otherwise, it scales the offset to the position and assigns a pitch deviation of 0.

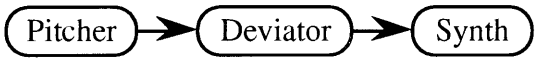



## 4.5. Pitcher

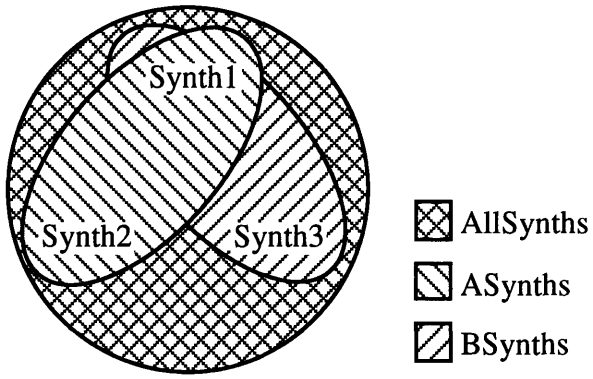
The Pitcher example exercises the CypherPitcher component in a short Cypher chain. Again, only the bottom two sensors are used. The connections for the bottom left sensor are diagrammed below. It is essentially a gesture component network for presence, stillness, flutter, and activity. The gesture values are accessed by components elsewhere. An identical gesture network exists for the bottom right sensor.



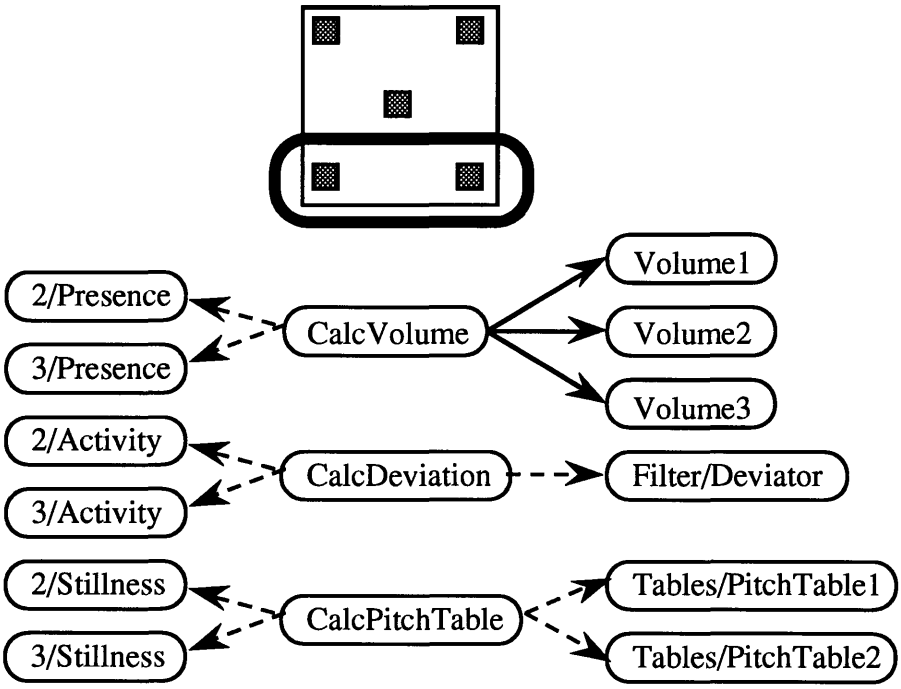

The Cypher chain with the CypherPitcher component is diagrammed below. Pitcher starts when the example does. It plays two notes with a velocity of 50 and a legato of 80% every 12 centiseconds. If there is a reference to a pitch table, Pitcher treats it as a probability table to generate pitches, otherwise MIDI note number 60 is used. Deviator, a CypherDeviator, initially does not deviate anything. Other components, described later, change its velocity deviation value.




Synth is actually a CypherPluggableFilter instead of a CypherSynth. A sub directory named Synths contains three CypherSynths, titled Synth1, Synth2, and Synth3, that play notes on three different MIDI channel. In addition, there are three component collections, named AllSynths, ASynths, and BSynths, that reference the synths. The figure below illustrates the membership of each of the collections. The pluggable filter Synth decides which synths receive Cypher events based on the amount of flutter in the bottom sensors. If there is insufficient flutter in both sensors, Synth1 plays all the events. Mutually exclusive flutter in sensor two or sensor three selects a synth collections, either ASynths or BSynths respectively. Synth distributes the events to the members of the selected collection based on a probability table called TableFor2. Enthusiastic fluttering in both sensors selects the AllSynths collection and Synth distributes the events based on a probability table named TableFor3.




Finally, we turn to the components that map gesture values to parameter changes. These components are characterized by the way they incorporate corresponding gesture values from both sensors in their determination of parameter values. The connections are diagrammed below. Dashed lines indicate references to components.




CalcVolume, a MaxPluggableValueObject, references the presence components to determine the greater presence value. That value is converted to MIDI volume messages by the three volume components, which correspond to the three synths in the Synths directory. CalcDeviation uses the greater activity value to scale the velocity deviation of Deviator in the Cypher chain. CalcPitchTable sets the pitch table of Pitcher. If there is very little presence in both sensors, it sets the pitch table to nil so that a single pitch is used

by Pitcher to generate notes. If there is presence in both sensors, PitchTable1 becomes the pitch table. However, if there is a lot of stillness on a low medium value, CalcPitchTable shifts the harmony by setting the pitch table to PitchTable2.

# 5 Evaluation

There are two aspects of this work that need to be evaluated: the software environment for designing responsive sound surfaces and the actual examples themselves. Unfortunately, there are no absolute metrics for evaluating either aspect. However, one measure of success for the software environment is how reusable and extensible it is. That issue must be addressed in light of the fact that at least two kinds of design processes will unfold and often merge within the environment. The first one is that of building a specific system for a particular responsive sound surface while the other seeks to extend the environment to provide a broader selection of components and resources to include in specific systems.

The designer of a specific system is primarily interested in reusable components. The components that currently exist within the environment are fundamental building blocks that must be interconnected and cross-referenced in order for them to do anything useful. The components are at a fundamental level so that the designer may mix and match to build a custom system that fits a particular context. However, the components are not at such a low-level that one would need an army of components to determine, say, the amount of activity on a sensor. Although I decided on an initial set of gesture components and a protocol specifying how to connect and access them, nothing in their design rigidly determines how they are to be used. In the responsive sound surface examples, they work with components that map their values to MIDI controller messages and with other components that convert their values to Cypher parameters.

The designer, who wants to extend the environment with new components that are intended to be used in many specific systems, is interested in how extensible the environment is: in particular, what must be done to create a component class that coordinates its actions with other components and the framework, archives itself to a file, and offers at least a form filling graphical interface. A new component that is a subclass of LizComponent or one of its descendants inherits all the necessary behavior for coordinating with other components and the framework, although some specialization is often required. It will also inherit mechanisms for archiving itself to a file that handle intrinsic and symbolic component references properly and in such a way that it need only be concerned about saving data specific to its class. Finally, the RSS development environment coupled with VisualWorks

offers abundant support for building forms filling component interfaces swiftly and visually.

Quite often, the two design processes overlap. While building a certain system, the designer may discover the need for a component that does not exist. If the missing component is very specific to the situation, it might not make sense to create a new component class that clutters up the environment with unreusable classes. Although the process for creating a new component class is streamlined, it does take a bit of time and might be disruptive for little bits of code. By taking advantage of a dynamic language's ability to compile code on the fly, LizPluggableComponent and its descendants fills the need for quick, one of a kind components.

A small pool of participants interacted with the various responsive sound surface examples. It was clear from their feedback that simple mappings, especially when layered on a sensor, could easily produce a complexity that belied the simplicity of its parts. Even the straightforward mappings of AmpMod, the simplest example, were not easily discerned as being as simple as they were due to the complexity of the sounds and the dynamics of the "sweet spot." Indeed, when I explained the mappings of AmpMod and how the sounds were synthesized to one participant, he jokingly accused me of cheating because the elements were so simple. Yet, after much previous experimentation, he had been unable to map out exactly what was going on.

In designing the examples, I was not interested in a detailed, instrument-like level of control, where the participant would know how to precisely control every aspect of the system. Indeed, one of the guiding principles was that a participant should have no particular set of skills or knowledge to interact with the system. As expected, a human participant naturally produced an unpredictable and dynamic environment, and much of the complexity of the system's responses could be derived from the rich behaviors of the participants while keeping the mappings of the system relatively straightforward. I tried to avoid the more blatant and obvious mappings of, say, position to pitch, but discovered that even mappings with only slightly more sophistication, especially when layered onto a sensor, could bring the participant easily to the point of confusion.

Some participants expressed a need to know exactly what was going on so that they could have instrument-like control over the system. Once the mappings were explained to them, they grew less interested in the system because the mappings were simple and the need for

exploration was removed. In the Ancient example, one person wanted the elements that built up and decayed in volume to do so more quickly, so that it would be obvious. Unfortunately, the obvious effect would have been mechanical and tedious.

Clearly, more work is needed in the area of how to anchor a participant without becoming annoyingly obvious or heavy. From my observations, a bare surface without embellishment and no other means of visual feedback is too disorienting. People had great difficulty finding the entire range of the sensors. As a result, I feel some sort of visual feedback indicating that a hand is within the range of a sensor is appropriate here, especially because it would not clog up the audio activity.

As an experiment, I had people try the system while viewing monitor windows that graphed the sensor values. One person remarked that he found it easier to watch and control the graphs before associating various gestures with changes in the sounds and textures. Overall, the monitors garnered too much attention and were too literal as a visual interface. People often remained transfixed by them even after they had acquired a good sense of the sensor's range and the various gestures. In the context of my examples, a better balance would be to have some visual elements that simply indicated whether a hand was in range and did not attract too much attention, yet were pleasing to the eye.

I feel the gesture components offer a sufficient number and an appropriate collection of gestures for the context of responsive sound surfaces. The number of effective gestures grows large when one factors in the partitioning of the sensor range, the layering of gestures at different time scales on an individual sensor, and the correlation of gestures in different sensors. The gestures that build up and decay over time really help to move away from a trigger sensibility where a simple action always has a similar and direct result. However, a gesture like flutter, which is simple to do once you know of it, might never be discovered. The intent behind flutter was to add a more subtle layer of interaction that would slowly be discovered over time as a reward for exploration. However, one might argue that flutter would probably not be discovered unless the interactions somehow lead a participant up to its discovery.

Even with straightforward mappings, it is simple to create a system where the participant would probably not discover most of the mappings. This would be even more so in an environment with sculptural surfaces and hidden sensors. The issue here is how the system can encourage participants to become more sophisticated and exercise a significant

number of the gestures at their disposal. Recent work by Maes et. al. on the ALIVE system [Maes et. al.94] includes coaching agents, which are weaved right into the interactive flow, that help a participant explore all the interactions of the system. Within the context of responsive sound surfaces, one possibility is a layer of components that profiles participants and adapts the parameters of the gesture components to their profiles over time. For example, if the participant has built up a lot of presence in a sensor the system might, dampen the more immediate and obvious interactions to highlight the more indirect ones. If the participant continues to overlook the less obvious interactions, make them more sensitive by adjusting the parameters of the relevant gesture components. Over time, we would expect a participant to become more aware of the ways of the system, and, by the same token, the system would adapt and personalize the experience for the participant.

The ability to customize LizNodeSentry with dynamically created blocks of code makes it a suitable mechanism for implementing a wide variety of node transition policies. The sentries in the examples are on the simple side and experience with systems containing more nodes and many different sentry policies would be needed to more fully evaluate LizNodeSentry. LizNodeSentrys' explicit links to nodes may become unwieldy in large systems and extending LizNodeSentry to encompass the more powerful description-based approach in Evans' work [Evans94] would alleviate that problem.

# 6 Future Directions

## 6.1. Implications of More Sophisticated Gesture Analysis

This work analyzed the features of simple gestures made in relationship to a location on a surface. Addressing the more general problem of analyzing gestures in free floating space would greatly enrich the vocabulary of physical gestures. It would also be interesting to include gestures based on the distance between two hands. However, care would be needed to ensure that all the gestures required no instruction or coaching.

With reliable and accurate x, y, and z hand coordinates, the number and shape of the sensitive areas on the surface would not be constrained by the geometry and number of sensors. In addition, the areas need no longer be static, but could dynamically change size and shape over time.

More sophisticated gesture analysis would certainly place greater demands on the system to the point where a separate gesture server might be necessary. In that case, the RSS development environment's distributed architecture and ability to communicate over TCP/IP would ease the integration of a gesture server.

## 6.2. Integrating Other Domains

In order to focus on the hand and sound interactions, I only experimented with a surface that was a bare, flat, and square sheet of Plexiglas. A sculptor might employ textured surfaces of various shapes and sizes or use the ideas underlying responsive sound surfaces to animate the space around sculptural elements.

Continuing into the visual domain, obvious extensions to the RSS development environment include video and graphic components. It is in these areas that the ground work for laying the RSS development on a distributed system really pays off. By creating separate video and graphic servers that communicate over TCP/IP, integrating them into the RSS development environment is straightforward. Video and graphic processing are generally computationally intense and a distributed architecture would prevent that

demanding processing from interfering with the performance of the rest of the RSS development environment.

## 6.3. Interactive Music for Video Games

Video game designers have been at the forefront of interactive systems implementation. A feat made all the more astonishing when one considers how cheap the hardware must be for video games. One perceptual trick of the trade is that good quality audio enhances the quality of mediocre graphics hardware. And so, video games generally have a musical score and sound effects. As video games begin to explore the nonlinear models of interactions, signaled by the emergence of many recent CD-ROM games, music that is pre-stored and therefore static becomes inappropriate and ultimately detrimental to a dynamic, interactive experience.

Alexander Rigopulos proposes musical "seeds" to generate music in real-time as one solution to the problem [Rigopulos94]. His seeds are short fragments of fundamental musical material including rhythmic, harmonic, melodic, and textural information. A group of idiomatically neutral analyzers extract parameters sets that represent various musical features of the seeds. Subsequent music, bearing characteristic properties of the original seed, is generated as a function of the parameter sets. By modifying any combination of the parameters over time, the unfolding music may elastically respond to external conditions.

Coupling a responsive sound surface with Rigopulos' seed-generated music system would be a powerful combination. By employing the gesture components that build up and decay over time, the parametric controlled music would smoothly adapt itself to an overall sense of what the participant is doing rather than jumping in fits and starts to every movement a participant makes.

Going further, it is easy to conceive of the idea of a higher-level musical instrument. One that would incorporate non-musicians in a musical activity. Many commercial keyboards with auto-music functions, overwhelm the musical initiate with a plethora of buttons and switches that are each tied to simple, musical changes. An alternative approach would be to do away with the buttons and switches in favor of a responsive sound surface where one could navigate through musical material and transformations spatially. The layering of interactions on individual sensors would create a rich environment for music making.

## 6.4. Incorporating Group Interactions

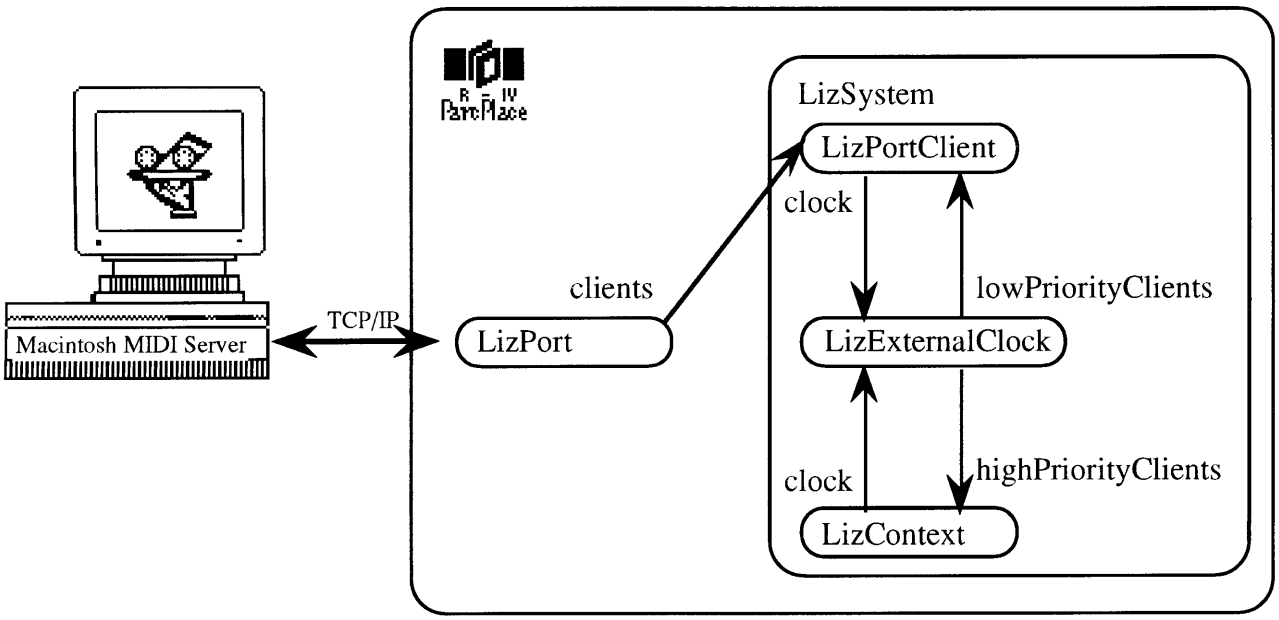
Perhaps the most interesting and challenging future direction for responsive sound surfaces is how to incorporate them within group interactions. At the technical level, this would certainly involve fleshing out the distributed system with many more computers. On a more conceptual level, one of the critical question is how to create a responsive space where the interplay of actions between people and the space dynamically shapes an experience that expands consciousness and perception while touching each participant in personal, individual ways. How does one unify the actions of a group of people with no particular set of beliefs, skills, and knowledge, and yet retain an individual sense of relevance and expression?

The *Brain Opera* [Machover94b] signals the emergence of a new art form, where people gather to participate in a shared, mediated experience and their actions have heightened significance. Our modern civilizations have lost the ancient, sacred and magical spaces where people gathered to commune with spirits and achieve an altered state of consciousness. It is my conviction that very large participatory interactive systems that are artistically conceived have the capacity to recapture some of the spirit of those faded magical experiences. It is also my hope that this work starts to pave the path toward very large participatory interactive systems that artists may build upon in their pursuit of these lofty aspirations.

# 7 Appendix: Implementation Details

## 7.1. Coordinating With the MIDI Server

A group of components works together to coordinate activities with the MIDI server to provide timing, MIDI input and output, and scheduling services. These include the LizPortClient, LizExternalClock, and LizContext. The diagram below illustrates the relevant relationships among the components.




LizPort is the object that establishes a connection with the MIDI server. It maintains a list of clients that are interested in MIDI data from the MIDI server. Although LizPortClients are generally its clients, LizPort does not care who they are as long as they adhere to its client protocol. LizPort sets up an input process that waits for MIDI packets from the MIDI server. When it receives MIDI packets, the input process determines whether the packet is a real-time or channel packet and sends the message **port:didReadSystemRealTimePacket:** or **port:didReadChannelPacket:** respectively to the clients along with the received packet. LizPort also has a buffer for MIDI packets that are to be sent to the server. It is important to buffer the output MIDI packets and then coalesce them into one large packet because sending packets over the network is expensive.

In the diagram above, LizPort has one client, namely LizPortClient. LizPortClient has references to LizPort and LizExternalClock and maintains an input queue of MIDI packets and a set of input handlers that want MIDI packets. When it receives the message **port:didReadChannelPacket:** from LizPort, LizPortClient simply sticks the MIDI packet onto its input queue. We buffer the input packets because the input handlers may output MIDI packets when they process the input packets. We want to handle the input packets all together and at a known time so that LizPort's output buffer will be flushed properly and the MIDI packets actually sent to the MIDI server.

LizExternalClock keeps a list of high priority and low priority clients. In general, high priority clients would generate MIDI output packets and low priority clients would ensure that the packets are actually sent to the MIDI server. In the diagram above, LizExternalClock has LizContext as a high priority client and LizPortClient as a low priority one. All clients, regardless of priority, must respond to the messages **clock:willTick:** and **clock:didTick:**. If the MIDI packet, sent by LizPort through the message **port:didReadSystemRealTimePacket:**, is a MIDI clock message, LizPortClient sends **clock:didTick:** with the time stamp of the packet to LizExternalClock, who expects to be told when an external clock ticks. LizExternalClock has a clock process that waits on a semaphore for external clock ticks. When it receives the **clock:didTick:**, LizExternalClock signals its semaphore so that its clock process may continue. The following bit of pseudo code reveals how the clock process proceeds.

send **clock:willTick:** to the high priority clients
send **clock:willTick:** to the low priority clients
set time to the real time
send **clock:didTick:** to the high priority clients
send **clock:didTick:** to the low priority clients

LizPortClient, which is a low priority client, responds to **clock:willTick:** by passing on the MIDI packets in its input queue to the appropriate handlers and responds to **clock:didTick:** by flushing LizPort so that any buffered MIDI packets are sent to the MIDI server.

LizContext references a clock and maintains a local scheduler, which is based on an implementation by Lee Boynton [Boynton87]. It provides a local context for scheduling tasks. The scheduler advance of LizContext determines how far ahead of real time the

context will schedule events. Scheduling into the future helps alleviate latencies due to system load, but increases the system's reaction time. Joe Chung's advice regarding the value of scheduler advance is particularly apt: "The rule of thumb is to use a scheduler advance that makes things sound better, but doesn't noticeably decrease responsiveness. [Chung91]" The ability to have multiple LizContexts increases the programmer's flexibility in meeting accuracy versus responsiveness requirements.

In the diagram above, LizContext is a high priority client of LizExternalClock. LizContext ignores the **clock:willTick:** message from LizExternalClock and responds to **clock:didTick:** by executing all the scheduled tasks that are ready.

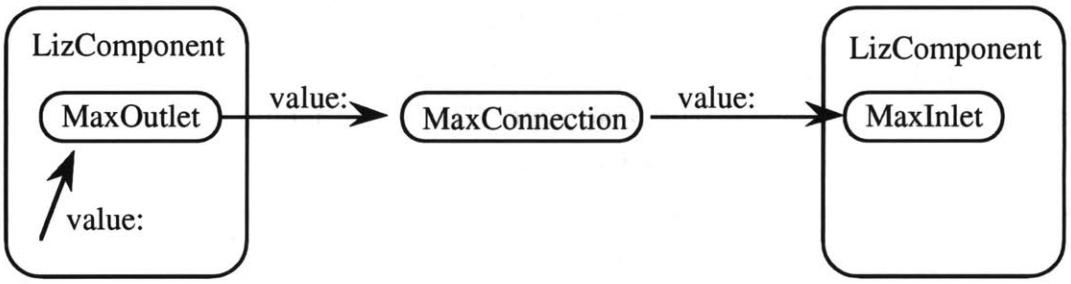
## 7.2. Archiving

To archive itself to a binary file, a LizComponent must specialize the method **addInitMessageTo:forBinaryWriter:**. Within that method, a particular class need only create a message with arguments that will perform initialization specific to the class. It should also give its superclass an opportunity to do the same by invoking **addInitMessageTo:forBinaryWriter:** on it. This scheme simplifies the archiving of new LizComponents; each class need only concern itself with initialization particular to itself and not worry about how its superclasses must initialize themselves.

After a component is read back in, the archiving mechanism sends it the message **awakeFromStorage**. At that time, a component tries to resolve each symbolic reference to an actual component or set it to nil if the object did not exist.

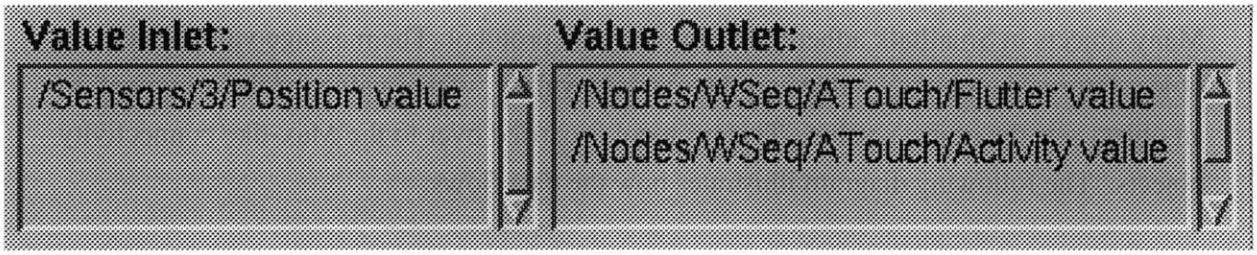
## 7.3. Max-Like Connectivity

Many objects in Smalltalk adhere to a simple value protocol: The message **value** sent to an object returns the object's value and the message **value:**, with a value for the argument, to an object sets the object's value. This protocol is carried over to the Max-like connectivity objects. The following diagram illustrates how the data flows:




MaxInlet has a block of code called *valueBlock* that handles the **value:** message. The LizComponent, who owns the MaxInlet, can either supply the value block or have the MaxInlet generate a block of code that will notify the LizComponent, using a message chosen by the LizComponent, when a new value arrives. MaxValueObject comes pre-packaged with one MaxInlet and one MaxOutlet. To handle received values, the programmer would only have to specialize the **valueInletValue:** method; MaxValueObject initializes its MaxInlet appropriately. MaxPluggableValueObject, a descendent of LizPluggableComponent, also comes pre-packaged with one MaxInlet and one MaxOutlet. In keeping with the spirit of pluggable objects, the designer may, at any time, submit blocks of code to handle input values.

Components with inlets and outlets have a simple textual interface, like the one below, for making connections on the fly. Two pieces of information are necessary to establish a connection in this interface: a component and a message to access the appropriate inlet or outlet of the component. In the text pane of value inlet below, "/Sensors/3/Position" is a source component and "value" is the root of the message sent to that source component to acquire the outlet for the connection. The actual message sent is the root appended with "Outlet" to produce **valueOutlet**. A similar procedure occurs for outlet connections except that the root of the message is appended with "Inlet."



## 7.4. MaxTable

The MaxTable component provides a way to store and edit an array of numbers. It is similar to the table object found in Max. In particular, it implements a **quantile** method that treats the array as a probability table. The **quantile** method calculates a y value by multiplying the sum of the array values by a random number between 0 and 1. Then, it iterates through the array to return the index of the first value that is greater or equal to the y value. In effect, each index occupies a portion of the sum of array values that is proportional to the "weight" or value at each index. Repeatedly invoking **quantile** returns each index at a frequency approximately proportional to the corresponding value. By treating the indices as MIDI note numbers, one may set up a simple pitch probability table with MaxTable and **quantile**.

## 7.5. RssSensor and RssSmoother

RssSensor, a descendent of LizComponent, references a LizExternalClock and a LizContext and has a value outlet. At set up time, it attaches itself to its LizPortClient as an input handler on a particular MIDI channel. LizPortClient will pass MIDI packets to RssSensor by sending the message **handleMidiChannelPacket:** with the MIDI packet as the arguments. RssSensor ignores all MIDI packets except for pitch bend, which it normalizes to values between 0 and 1. At every clock tick of its LizContext, RssSensor sends the last normalized pitch bend value through its outlet.

To find the position of a sensor, one could simply take RssSensor's value. However, connecting RssSensor to RssSmoother and treating the value of RssSmoother as the position will reduce jitter and minimize the effect of dropped sensor data packets. RssSmoother, like most of the gesture components, is a descendent of MaxValueObject and so it has a MaxInlet and a MaxOutlet. Its sole parameter is a smoothing factor that should be a number between 0 and 1 inclusive. The smoothed value is calculated by the following formula:

smoothedValue = smoothingFactor * aValue + (1 - smoothingFactor) * previousValue

# 8 Bibliography


[Allik & Mulder93] Kristi Allik and Robert Mulder, *Skyharp: An Interactive Electroacoustic Instrument*, Leonardo Music Journal, Vol. 3, pp. 3-10, 1993.

[Baker88] Kenneth Baker, *Art for the Ears at Capp Street Project*, San Francisco Chronicle, January 1, 1988.

[Behrman & DeMarinis82] David Behrman and Paul DeMarinis, *Sound Fountain*, project description, 1982.

[Behrman & Lewis87] David Behrman and George Lewis, *A Map of the Known World*, project description, 1987.

[Boynton87] Lee Boynton, *Scheduling as Applied to Musical Processing*, MIT Media Laboratory report, November, 1987.

[Buchen92] Bill and Mary Buchen, *Sound Engine*, project description, 1992.

[Bunish94] Christine Bunish, *A Perfect Fit Interactive Multimedia Naturally Suited To Museums*, In Motion, April, 1994.

[Caudell93] Robin Caudell, *Sound and light 'sculpture' interacts in Myers Gallery*, The Arts, September 24, 1993.

[Chadabe89] Joel Chadabe, *Interactive Composing: An Overview*, The Music Machine, edited by Curtis Roads, The MIT Press, Cambridge, Massachusetts, 1989.

[Chung91] Joseph Chung, *Hyperlisp Reference Manual*, available from the MIT Media Laboratory, Massachusetts Institute of Technology, 1991.

[Cox and Novobilski91] Brad Cox and Andrew Novobilski, *Object-Oriented Programming*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[Daniel94] Sharon Daniel, *Strange Attraction: Non-Logical Phase-Lock over Space-Like Dimensions*, artist statement, 1994.

[Dannenberg89] Roger Dannenberg, *Real-Time Scheduling and Computer Accompaniment*, Current Directions in Computer Music Research, edited by Max V. Matthews and John R. Pierce, pp. 223-261, MIT Press, Cambridge, 1989.

[Downton91] Andy Downton and Graham Leedham, *Human aspects of human-computer interaction*, 13-27, Engineering the Human-Computer Interface, edited by Andy Downton, McGraw-Hill Book Company (UK) Limited, 1991.

[Evans94] Ryan Evans, *LogBoy Meets FilterGirl A Toolkit for Multivariant Movies*, M.S. for MIT Media Laboratory, 1994.

[Furlong88] Lucinda Furlong, *ELECTRONIC BACKTALK The Art of Interactive Video*, The Independent, May, 1988.

[Gann92] Kyle Gann, *Soundscapes R Us*, The Village Voice, May. 20, 1992.

[Gann93] Kyle Gann, *Handmade in Holland*, The Village Voice, Nov. 30, 1993.

[Gassée90] Jean-Louis Gassée, *The Evolution of Thinking Tools*, The Art of Human-Computer Interface Design, edited by Brenda Laurel, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Gershenfeld94] Neil Gershenfeld et. al., *On The Electrodynamics of Bodies*, preprint, MIT Media Laboratory, 1994.

[Goldberg & Robson89] Adele Goldberg and David Robson, *Smalltalk-80: the Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Houbart93] Gilberte Houbart, *"It was a knowledge war": an Evolving documentary*, M.S. thesis proposal for MIT Media Laboratory, 1993.

[Janney79] Christopher Janney, *Soundstair*, project description, 1979.

[Janney88] Christopher Janney, *Reach!*, project description, 1988.

[Janney94] Christopher Janney, *Biographical Statement*, 1994.

[Kerckhove91] Derrick de Kerckhove, *A Very Nervous Piece*, Leonardo, 24:2, pp. 131-135, 1991.

[Korg92] *Korg Wavestation SR Player's Guide*, Korg Inc., 1992.

[Krueger91] Myron Krueger, *Artificial Reality* , Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[Laurel93] Brenda Laurel, *Computers as Theater*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[Machover92] Tod Machover, *Hyperinstruments: A Progress Report*, available from the MIT Media Laboratory, 1992.

[Machover94a] Tod Machover, *Brain Opera Project Description*, available from the MIT Media Laboratory, 1994.

[Machover94b] Tod Machover, *Brain Opera Proposal for Atlanta Olympics*, available from the MIT Media Laboratory, 1994.

[Maes et. al 94] Pattie Maes, Trevor Darrell, Bruce Blumberg, Sandy Pentland, *The ALIVE System: Full-body Interaction with Animated Autonomous Agents*, submitted to SIGGRAPH '94.

[Matsumoto93] Fumiaki Matsumoto, *Using Simple Controls to Manipulate Complex Objects: Application to the Drum-Boy Interactive Percussion System*, M.S. for MIT Media Laboratory, 1993.

[Meyer92] Bertrand Meyer, *The New Culture of Software Development: Reflections on the practice of object-oriented design*, from Advances in Object-Oriented Software Engineering, edited by Dino Mandrioli and Bertrand Meyer, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Morgenroth93] Lee Morgenroth, *Movie, Talkies, Thinkies: Entertainment When the Moral is an Experience,* MS thesis proposal for MIT Media Laboratory, 1993.

[NeXT92] NeXT, *NeXTSTEP General Reference Vol. 1*, NeXT, 1992.

[Nyman74] Michael Nyman, *Experimental Music*, Schirmer Books, New York, New York, 1974.

[Oliveira et. al 94] Nicolas de Oliveira, Nicola Oxley, and Michael Petry with texts by Michael Archer, *Installation Art*, Smithsonian Institution Press, 1994.

[ParcPlace92a] *Objectworks\Smalltalk User's Guide*, ParcPlace Systems, 1992.

[ParcPlace92b] *VisualWorks User's Guide*, ParcPlace Systems, 1992.

[Phillips88] Liz Phillips, *Graphic Ground*, artist statement.

[Phillips91] Liz Phillips, *Mer Sonic Illuminations*, artist statement.

[Pope92] Stephen Travis Pope, *The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers*, Computer Music Journal, 16:3, pp. 73-91, 1992.

[Reismann88] David Reisman, *Telescreen as text Interactive Videodisc Now*, Artscribe International, May, 1988.

[Reynolds93] Gretchen Reynolds, *Imaging at the Museum*, Wired, December, 1993.

[Rigopulos94] Alexander Rigopulos, *Growing Music from Seeds: Parametric Generation and Control of Seed-Based Music for Interactive Composition and Performance*, M.S. for MIT Media Laboratory, 1994.

[Rockwell81] John Rockwell, *Avant-Garde: Liz Phillips Sound*, The New York Times, May 14, 1981.

[Rokeby91] David Rokeby, *Very Nervous System*, artist statement.

[Rowe91] Robert Rowe, *Machine Listening and Composing: Making Sense of Music with Cooperating Real-Time Agents*, Ph D thesis for MIT Media Lab, 1991.

[Rowe93] Robert Rowe, *Interactive Music Systems*, The MIT Press, Cambridge, Massachusetts, 1993.

[Shearing94] Graham Shearing, *Melodic trap of the Sonic Garden*, Pittsburgh Tribune-Review, June 17, 1994.

[Stevens90] W. Richard Stevens, *Unix Network Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Wishart85] Trevor Wishart, *On Sonic Art*, Imagineering Press, York, United Kingdom, 1985.

[Teitelbaum84] Richard Teitelbaum, *The Digital Piano and the Patch Control Language System*, Proceedings of the ICMC, pp. 213-216, 1984.

[Vercoe84] Barry Vercoe, *The Synthetic Performer in the Context of Live Performance*, Proceedings of the ICMC, pp. 199-200, 1984.

[Waisvisz et. al 94] Michel Waisvisz et. al, *STEIM de Zetgevooisde Bliksem*, Steim publication, 1994.

[Waxman et. al. 94] David Waxman, MariBeth Back, Jin Kang, Joshua Smith, Adam Lindsey, *Gesture Cube*, project description, 1994.

[Weinbren85] Grahme Weinbren, *The Limousine Authoring System*, unpublished paper, 1985.

[Yasui90] Todd Allan Yasui, *Soothing Sounds for Mean Streets*, The Washington Post, August 20, 1990.

[Zicarelli87] David Zicarelli, *M and Jam Factory*, Computer Music Journal, 11:4, pp. 13-29, 1987.

**Software:**

[Neeracher93] Matthias Neeracher, *GUSI - Grand Unified Socket Interface*, public domain software library, 1993.

[Puckette & Zicarelli90] Miller Puckette and David Zicarelli, *Max Development Package*, Opcode Systems, 1990.

[Zicarelli88] David Zicarelli, *M, The Interactive Composing and Performing System*, Intelligent Music, 1988.