SoundBlocks and SoundScratch: Tangible and Virtual Digital Sound Programming and Manipulation for Children

by
John Harrison

BM – Bachelor of Music
Cleveland Institute of Music
May 1993

Submitted to the Program of 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 2005

John Harrison
Program in Media Arts and Sciences

Barry Vercoe
Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Andrew B. Lippman
Chairman, Departmental committee on Graduate Studies
Program in Media Arts and Sciences

SoundBlocks and SoundScratch: Tangible and Virtual Digital Sound Programming and Manipulation for Children

John Harrison

BM – Bachelor's in Music
Cleveland Institute of Music
May 1993

Submitted to the Program of Media Arts and Sciences,
School of Architecture and Planning, on August 5th, 2005
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 2005

ABSTRACT

Creative Digital sound manipulation is a powerful means of personal expression. However, it remains explored by only a small number of engineers, mathematicians, and avant-garde musicians and composers. Others find the interfaces both obtuse and focused more on how the sounds are manipulated than what expressivity the manipulations offer. Yet digital sound manipulation can be accessible to everybody. It can even be a powerful way for people to explore, design, and create while learning about mathematics, dataflow, networks, and computer programming.

SoundBlocks and *SoundScratch* are two different environments in which children can manipulate digital sound. *SoundBlocks* is a tangible programming language for describing dataflow with adaptive, context-aware primitives and real-time sensing. *SoundScratch* is a set of sound primitives that extend the media-rich capabilities of the children's programming language called *Scratch*.

Both environments have been created and developed as a way to explore how it might be possible to construct an environment in which youth design their own sounds. Children ages 10-15 years old have explored the environments and participated in user studies. Music educators have observed these studies, and their observations are summarized.

Thesis Supervisor: Barry Vercoe
Professor of Media Arts and Sciences

SoundBlocks and SoundScratch: Tangible and Virtual Digital Sound Programming and Manipulation for Children

John Harrison

Thesis Reader:

Mitchel Resnick
Associate Professor of Media Arts and Sciences
MIT Media Laboratory

Thesis Reader:

Joseph Paradiso
Associate Professor of Media Arts and Sciences
MIT Media Laboratory

Acknowledgements

My experience at the Media Lab has been profound. Everybody I have met and have spoken with at the lab has contributed in some way. Below are a few of the people who come to mind as I write this. There are countless others who slip my mind and who deserve equal credit:

Both **Wichita State University,** and **The Wichita Symphony Orchestra** gave me extended leaves so I might pursue opportunities at the Media Lab. Thank you to the faculty and administration of these organizations that made this possible.

Josh Lifton has been supportive and encouraging ever since I met him at the lab. He has an unusual gift as a teacher, and he gave me the initial encouragement and support I needed when I was first learning Python.

Camilla Noergaard Jensen was very generous in sharing her ideas and her time. She has wonderful ideas both for her own projects and for *SoundBlocks*, and I learned a great deal from talking to her.

When I first came to the lab, I was a bit overwhelmed with the depth of sound synthesis and the possibilities for research exploration. **Brian Whitman** was wonderful in helping me understand these various areas and how I might research them further. He has always been extremely helpful as various issues with computers and research have arisen.

Victor Adán has been a great friend and a great supporter. As I worked through the various problems that came up in the project, Victor was always there to help me think through the issues and offer helpful suggestions. Moreover he has been a great friend through all of this, and I have thoroughly enjoyed the multiple cups of coffee we have drunk together.

Judy Brown always gave great feedback. **John DiFrancesco** continues to lose his mind while helping all of us in the Media Lab shop. **Oren Zuckerman,** by writing his Master's thesis *System Blocks: Learning about Systems Concepts through Hands-on Modeling and Simulation,* gave me a great model to study when writing this thesis. **David Gatenby** was great to share the driving of the wambulance as we both worked on our theses.

The entire **Hyperinstruments** group has been great: **Tristan Jehan**, **Roberto Aimi**, **David Sachs**, **Adam Boulanger**, **Diana Young**, and **Mary Farbood**, not to mention **Tod Machover** himself.

Richard Boulanger has been a great inspiration. Because he was generous enough to allow me to sit in on his Csound class at *Berklee*, I finally found the structure I needed to fully learn Csound from the inside out.

Brian Silverman, through his work and our conversations, has been a great inspiration. Brian's creativity and excitement in his work and his valuable feedback in talking about my work has been vital for me.

John Maloney, besides being an ace programmer and extremely intelligent individual, is one of the most generous and kindest people I have ever met. His willingness to work with me, leading me through learning *Squeak* and finding my way in his *Scratch* code was well beyond the call of duty.

Neil Gershenfeld gave me a solid foundation on a broad range of topics in his *How To Make Almost Anything* course. Moreover, his method for structuring the course was an inspiration.

Joe Paradiso has been a great inspiration for my work at the lab from the start. Initially his *Sensors for Interactive Environments* course gave me a great introduction to electronics, sensors, what had been done and what was possible in this area. Moreover, Joe's research with sound and music captured my imagination. His analog synthesizer was also a great source of inspiration.

Mitchel Resnick has also been a great source of inspiration ever since I learned of his work. His contributions to children's learning are profound and I hope to continue to be a part of the community that supports and contributes to his work.

Barry Vercoe has been central to all of my work. It would have been next to impossible to construct a sound synthesis environment without Barry Vercoe's contributions to the field. Without Barry Vercoe, Csound would not exist. Barry gave me the freedom to explore the lab entirely. He has supported me as both a musician and a researcher.

My family, especially my brother **David Harrison** and my mother **Trudy Harrison** have been there for me throughout. Since my initial ideas of coming to the Media Lab, they have encouraged me and given me the valuable feedback I needed.

During the evaluation phase, several music educators observed students and gave me valuable feedback. These included: **Ann Gregg**, **Michael Bonner**, **Nicolasa Kuster**, and **Susan Mayo**.

Of course, the projects themselves would have all been pointless without **the youth**. Thank you to each and every one of them, as well as to **David Cavallo**, **Amon Millner**, and **Marlon Orozco** who all made it possible for me to meet the youth.

Victor and I shared a UROP, **Ting-Hsiang Hwang**, who we called Tony. Tony helped with the *SoundBlocks* project, and built a few of the blocks.

One of the most rewarding aspects of the *SoundBlocks* project was the opportunity to work with **Andrew McPherson**. Andrew fully implemented the network protocol used in the blocks and physically built almost all of them. He wrote the initial mid-level Python code that realized the structure of the network and many of his ideas shaped the functions of the blocks. He continues to be an equal partner in the project. Andrew has been fantastic to work with. His ideas are amazing. When it comes to designing and building electronics, he is one of the fastest people I know.

My roommate and good friend of 20 years **Michael Bonner** proofread this thesis and was an invaluable sounding board as I discussed both *SoundBlocks* and *SoundScratch* with him. Michael inspires me in the way he leads his life.

Finally, none of this would have been possible if my wife **Nicolasa Kuster** had not encouraged me to pursue the opportunity to study at MIT. During my residency at MIT, Nic and I have endured a long-distance relationship. Through this she has been steadfast in her support for me and for my work here. Her selflessness in her support is a model for my own life. In Nic I have found my life partner.

SoundBlocks and SoundScratch: Tangible and Virtual Digital Sound Programming and Manipulation for Children

John Harrison

"My freedom will be so much the greater and more meaningful
the more narrowly I limit my field of action
and the more I surround myself with obstacles.

Whatever diminishes constraint diminishes strength.
The more constraints one imposes, the more one frees one's self
of the chains that shackle the spirit."

-Igor Stravinsky, *Poetics of Music*

This thesis is dedicated in memory of my father, **Burton Harrison** (1927-1997).

Table of Contents

Figures and Tables

1. Introduction

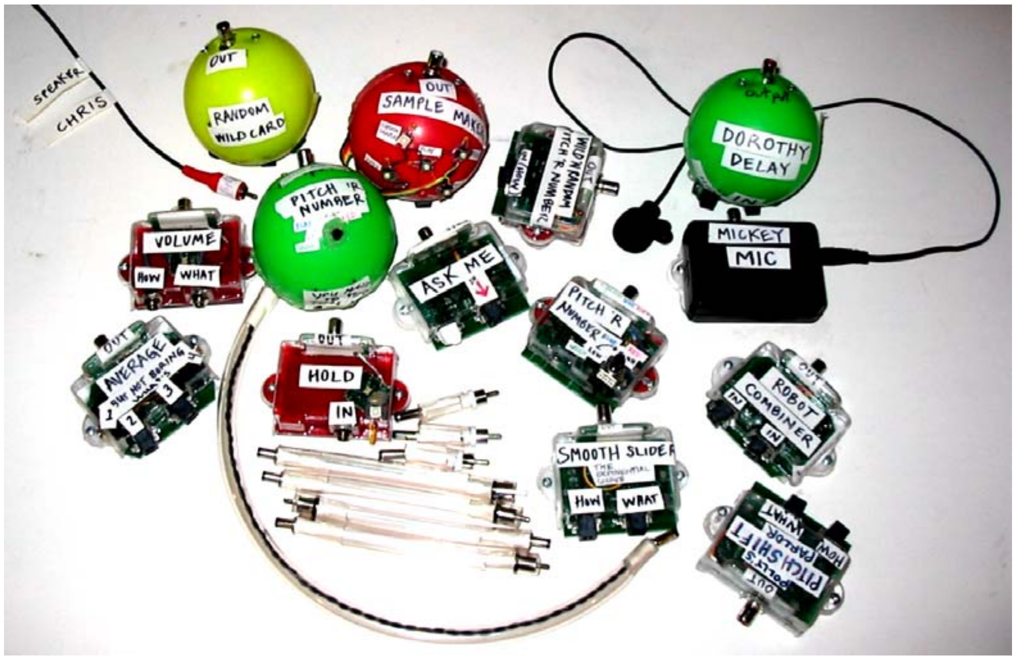
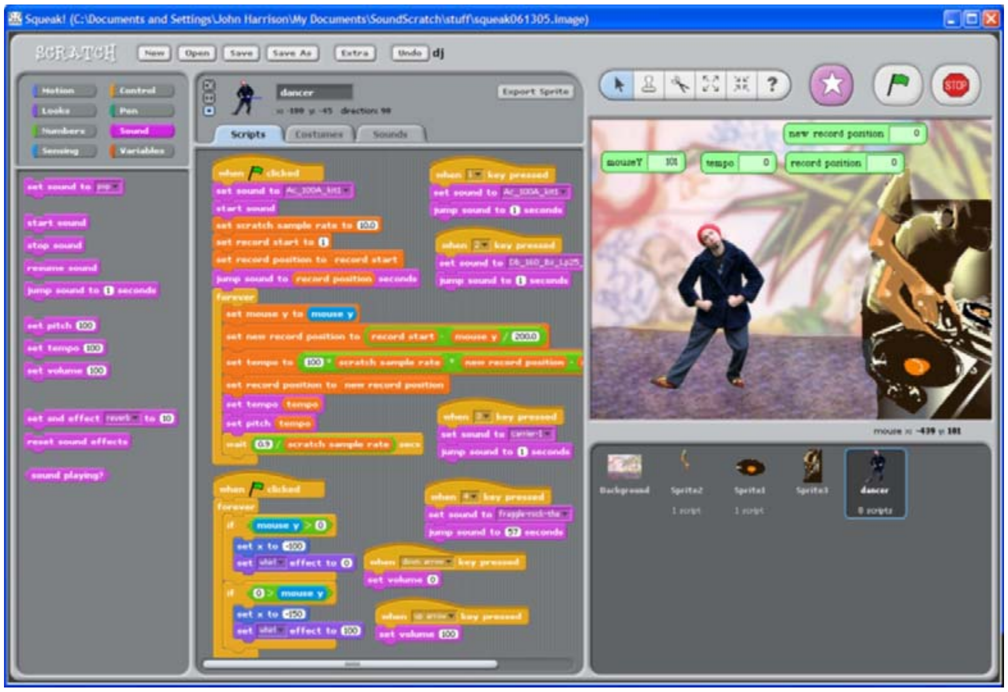


Figure 1: A Complete Set of *SoundBlocks*



```
when [flag] clicked
set sound to Ac_100A_kit1
start sound
set scratch sample rate to 10.0
set record start to 1
set record position to record start
jump sound to record position seconds
forever
    set mouse y to mouse y
    set new record position to record start - mouse y / 200.0
    set tempo to 100 * scratch sample rate * new record position -
    set record position to new record position
    set tempo tempo
    set pitch tempo
    wait 0.9 / scratch sample rate secs

when [flag] clicked
forever
    if mouse y > 0
        set x to -100
        set effect to 0
    if 0 > mouse y
        set x to -150
        set effect to 100

when 1 key pressed
set sound to Ac_100A_kit1
jump sound to 1 seconds

when 2 key pressed
set sound to DB_160_8z_1p25
jump sound to 1 seconds

when 3 key pressed
set sound to carrier1
jump sound to 1 seconds

when 4 key pressed
set sound to fraggle-rock-the
jump sound to 57 seconds

when down arrow key pressed
set volume 0

when up arrow key pressed
set volume 100
```

Figure 2: *SoundScratch* with *DJ Scratch*, an example program from Jay Silver

We are surrounded by digitally synthesized sounds. When we get in our cars, digital bells tell us our lights are on or our seat belts aren't fastened. We go to the movies, and the show is filled with a variety of sound effects. Almost all of our popular music has a large sampling of digital audio. It is commonplace for us to hear digital sound, and we accept it as part of our everyday world. Digitally manipulated sounds are as normal and natural to us as the natural environmental sounds that surround us.

For musical expression, digitally synthesized sounds offer a new dimension to explore. Any preexisting sound can be manipulated in countless ways with filters, re-samplers, vocoders, and more, to bring out previously unrecognized qualities in the sound. With the power of today's computers, these manipulations can happen in real time, extending the concept of a sound designer to include digital performer. In the words of composer Trevor Wishart, "with digital synthesis, we can now explore the multidimensional space of sound itself, which may be molded like a sculptural medium in any way we wish (Wishart 1994)."

Current digital synthesis interfaces use a combination of three techniques for interaction: mathematics, data manipulation, and physical or emulated hardware. Mathematical applications include *Csound* (Vercoe 1985) and *SuperCollider* (McCartney 1996). These languages give users the ability to describe the signals they wish to manipulate as a function of time and as a function of the signals operating on each other. They are excellent tools for understanding how sound is created mathematically, and for the physical modeling of sounds.

Data manipulation languages include *Pd* (Puckette 1997)*, Max/MSP* (D. Zicarelli, Yaylor et al. 1990-2004; Zicarelli, Yaylor et al. 1997-2004) and *VVVV*(Meso 1999-2005). These languages are graphical. Patch cords connect various objects and the relationships between these objects mold the sound through data flow. Graphical data manipulation interfaces are an evolution of the original hardware devices and the interfaces they provided.

Graphical interfaces are an artifact of how the original synthesizers and sequencers were designed. They are powerful in that they typically expose all states that a device can be in. No state for the device is hidden from the user. Commercial applications such as *Reason*

(Propellerhead 2001-2005) emulate these hardware devices, even going as far as to visually recreate the devices and patch cords, complete with stunning graphics.

Each of these approaches to shaping sound allows limitless possibilities for sound manipulation. However, viewed as tools for personal expression, they are indirect to the user, buried behind mathematics, networking, and/or gadgetry. While the tools succeed in giving users nearly limitless capabilities for sound manipulation, they fail in that the interface they provide can distract from the purpose of these tools in the first place. Users may quickly feel inundated, overwhelmed, or merely dismissive with a mathematical language for describing sound when what they are looking for is what expressive capabilities they might find in these manipulations and not the manipulations themselves. This has led many artists and musicians to feel that digital manipulation in media is irrelevant to their work.

In contrast to these approaches, we have created and tested *SoundBlocks* and *SoundScratch. SoundBlocks* is a tangible environment where young users connect blocks to describe network dataflow. *SoundScratch* are a set of extensions to manipulate audio in the children's programming language called *Scratch*. Both environments emphasize the expressive capabilities of the sounds through the act of creation and design. These environments are biased toward digital sound manipulation as a personal, meaningful and fun artistic endeavor, rather than as a venture into mathematical, electronic or networking relationships. Lead by their own curiosity, children can design their own sounds by exploring these environments. In doing so, they will indirectly learn a great deal about networks, mathematics and hardware synthesizers and sequencers. The environment will shift the child's focus from the *product* of creation to the *process* of creation.

2. Extended Examples

2.1 SoundBlocks

SoundBlocks is a set of 14 blocks that children use to manipulate live and their recorded sounds. Each block has a name intended to appeal to kids and also to give some intuition as to what the block does. The blocks each have one output, between 0 and 4 inputs, and an RGB LED, which can show various hues. Some blocks also have sensors: buttons, a knob, or a microphone.

Users attach the blocks to each other by selecting among the 9 semi-rigid cables of various lengths. The resulting network describes a set of sound manipulations.

Besides manipulating the blocks, a user can also interact with the system using the sensors. As the user connects and disconnects blocks, turns knobs, presses buttons or speaks into the microphone, he hears the results immediately. At all times the network creates sound as determined by the configuration and state of the network and sensors.

To the user, it appears that the blocks process the audio manipulations. Internally, however, no audio is processed in the blocks. Instead, the blocks function as a tangible interface to a computer which processes all of the sound manipulations. A full description of this process can be found in Chapter 5.

Below I lay out a set of example patches to offer some intuition as to how *SoundBlocks* function to the user.

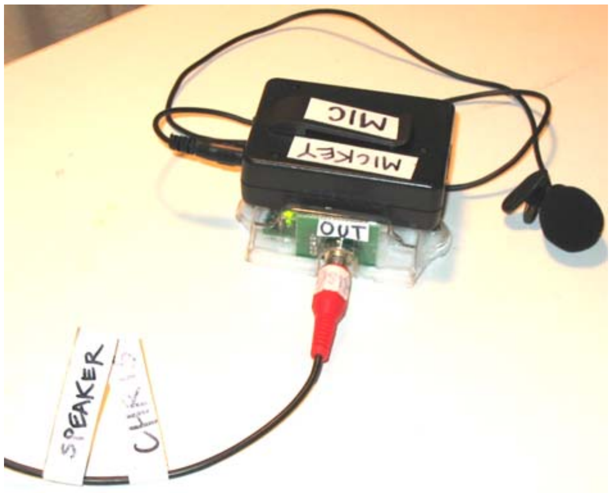


Figure 3: Direct Microphone Output

Chris the Speaker *is an RCA cable that attaches to a computer through a host circuit. The user hears whatever signal is symbolically sent to* Chris. *In this example,* Mickey Mic *is directly connected to* Chris, *so we hear the microphone signal.*

Besides one OUT, every block has an RGB LED, which gives users feedback related to the state of the block. Mickey Mic's *LED is to the left of his OUT jack and shows the amplitude of the signal.* SoundBlock *LEDs show ranges by changing shades of color from green, to blue, to red. The LED is green in this example, showing the amplitude of a signal is at a minimum i.e. the microphone is picking up no audio.*

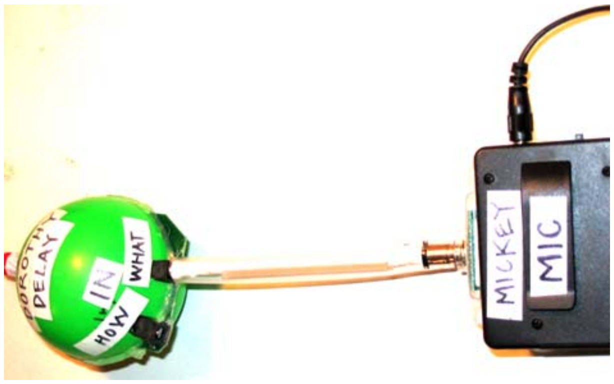


Figure 4: Microphone with 1.5 Second Delay

Mickey Mic's *"out" is connected to a block called* Dorothy Delay's Den *which is, in turn, connected to* Chris the Speaker. *The blocks named after either rooms or places are modifiers. They typically contain a "how" input and a "what" input: What do we want to delay? How much do we want to delay it? In this case, the "what" is Micky Mic and the "how" is unspecified. The system defaults to a delay of 1.5 seconds.*

Besides showing the state of blocks, LEDs blink in sequence to show signal flow in the network. In this example, the LEDs will blink from Mickey Mic *to* Dorothy Delay's Den*, showing that the signal flows from one to the other.*

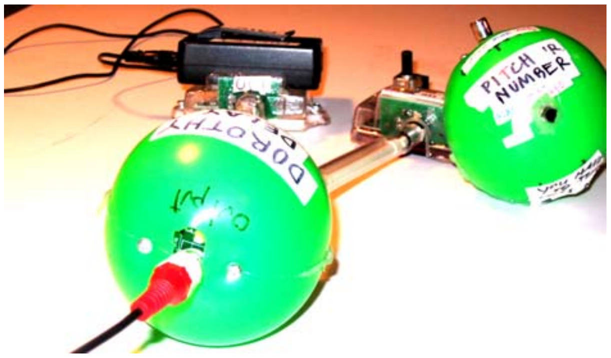


Figure 5: Microphone with Variable Delay

We add to the previous configuration Pitch 'R Number *(as in Pitch* ***or*** *Number), and connect it to* Dorothy Delay's Den's *"how" output. Now delay is controlled by the user with* Pitch 'R Number's *potentiometer --- a knob that the user controls.*

Two Pitch 'R Number blocks *are shown. They are both the same, except for the form factor and minor implementation details.*

By turning Pitch 'R Number's *knob while the microphone receives audio, the user shifts the pitch of the output by creating the Doppler effect. In this example,* Pitch 'R Number *is sending parameter values.*

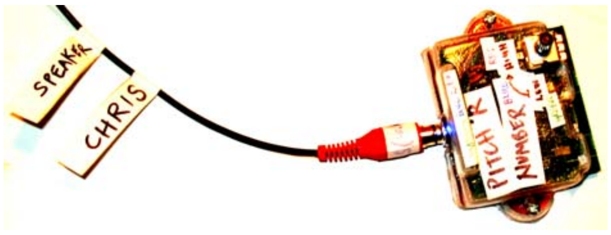


Figure 6: *Pitch 'R Number* as Audio

When connected directly to Chris*,* Pitch 'R Number *changes its audio output from numbers to tones. As the user turns the knob, a square wave shifts from 50Hz to 1kHz exponentially. In this picture, the LED is blue, so we know we are hearing a tone in the middle of this range.*

Note Pitch 'R number *has changed its output from numbers in Figure 5 to tones in this figure. This is an example of the context-aware adaptive behavior that all of the block demonstrate in the system.*

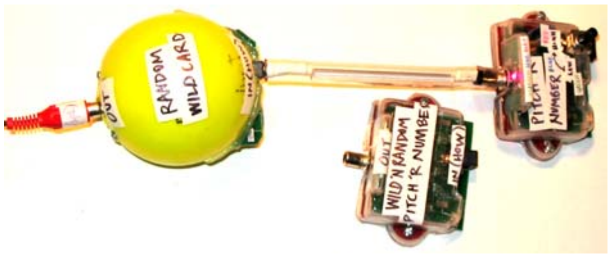


Figure 7: *Wild N' Random*

A block called Wild 'N Random Pitch 'R' Number *(a.k.a.* Random Wildcard*) sends random pitches or numbers at a steady speed. The speed is set by the* Wild 'N Random's *"how" input which is, in this case,* Pitch 'R Number*. Since* Pitch 'R Number's *LED is red and* Wild 'N Random *is connected to* Chris*, we know the user is hearing square waves of a fast and steady pace between 50Kz and 1khz.*

Two Wild 'N Random *blocks are shown. Both are the same except for form factor.*

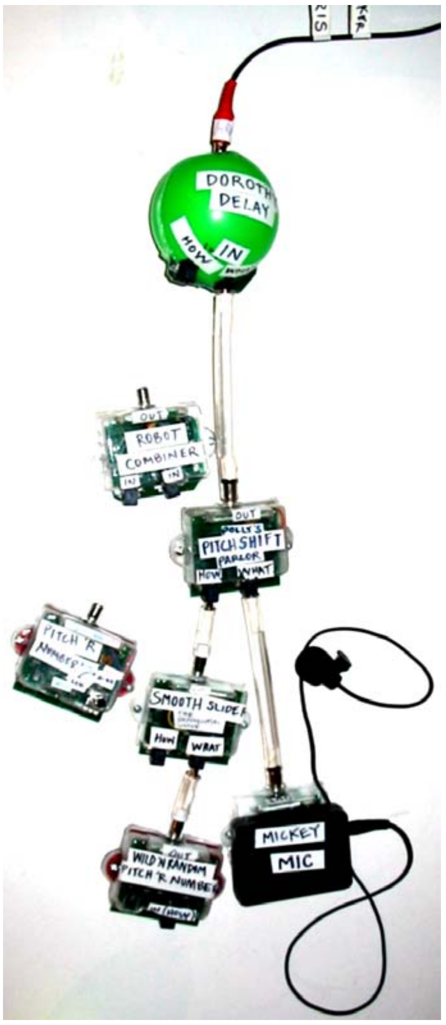


Figure 8: A PitchShifting Network

This configuration is a popular one with children. Smooth Slider *interpolates through values sent to its "what" input.* Polly's Pitchshift Parlor *receives these interpolated values in its "how" input and* Micky Mic *in its "what" input. Therefore,* Polly *pitch-shifts the microphone input as a function of the sliding values. The entire signal is delayed by* Dorothy Delay's Den *before being sent to* Chris *and thus being heard by the user.*

A related network is to substitute the Pitch 'R Number *block in place of the* Smooth Slider *and* Wild 'N Random *blocks. Then the user can control the pitch shift with the knob.*

The Robotic Combiner Diner *functions as a vocoder, and children often use it interchangeably with the pitch shifter.* Robotic Combiner Diner's *inputs are context-aware. In this example, the microphone will be assigned the formant signal and the* Smooth Slider *will be the carrier, regardless of which is connected to which input. If either or both inputs do not have blocks attached to them, the block will choose default carrier and/or formant signals.*

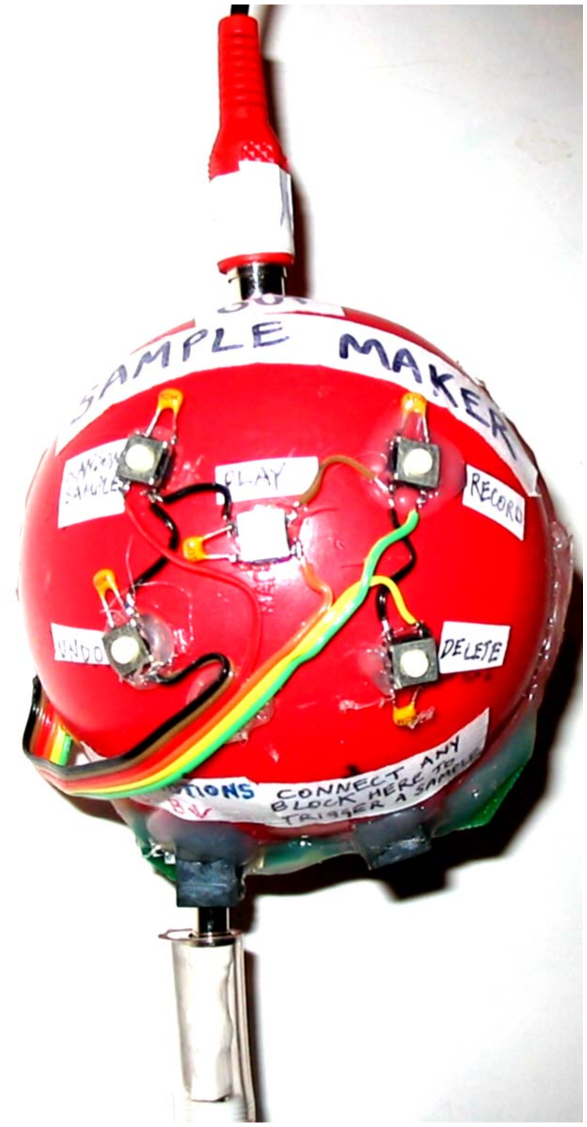


Figure 9: ***The Sample Maker***

The Sample Maker *maintains a list of samples recorded in the network. When connected to* Chris, *any combination of blocks in the network can be plugged into its "record" input to record a sample. Samples can be added, deleted, and reviewed.*

Besides recording their own samples, users can choose to add a sample from an internal list of pre-recorded samples. The pre-recorded samples are short and widely varied. They include sounds from traditional musical instruments, a horse galloping, whistles, and trains. If the user presses the "random sample" button, SoundBlocks *will randomly choose one of these pre-recorded samples and add it to the sample list*

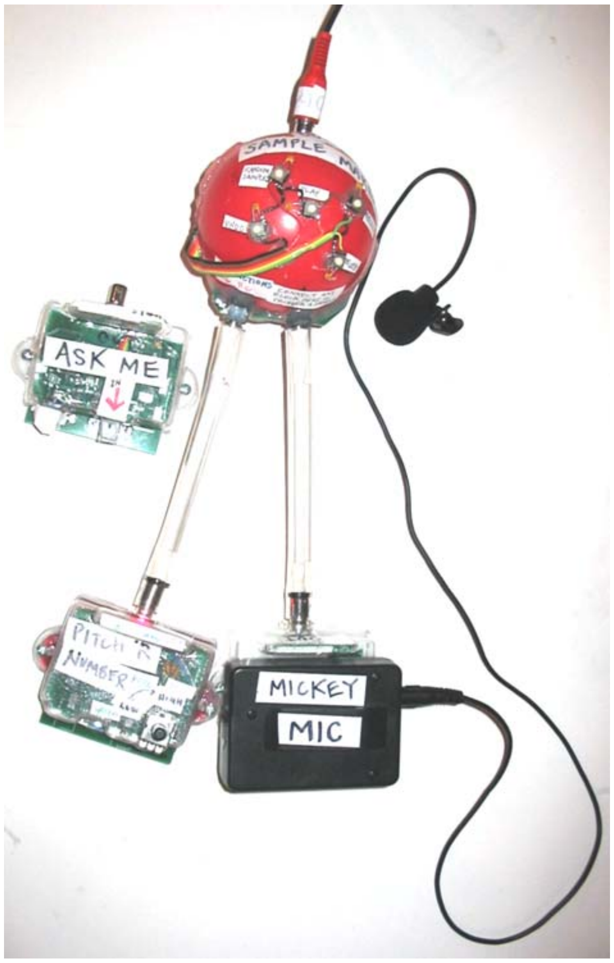


Figure 10: A simple Sample Maker Network

When functioning as a block in the network, The Sample Maker *has two inputs: "what" triggers what sample should be played. "How" controls the playback speed. In this example,* Micky Mic *is attached to* The Sample Maker's *"what" input, so the amplitude of the microphone is controlling what sample should be played.* Pitch 'R Number *is attached to* The Sample Maker's *"how" input, so its value is controlling the playback speed.*

The "Ask Me" *block is a helper block for users. When it is attached to* Chris *and a child block is plugged into it, it explains the child block's function and suggests possible networks for this block.*

2.2 SoundScratch

SoundScratch is the name I have given to the sound extensions I have added to a children's programming language called *Scratch*. In *Scratch*, users write scripts that manipulate costumes on a screen. The costumes can be any bit-mapped element, such as the user's own computer drawings or jpeg images. The default costume is a cat. To write scripts for these costumes, users drag interlocking blocks from the command palette on the left of the screen to the script window in the middle of the screen. They can then watch their scripts execute in the world pane, a white area in the upper-right-hand corner where the costumes interact.

In *SoundScratch*, users manipulate audio elements in much the same way they manipulate visual elements in *Scratch*. Below I lay out some example scripts showing the functionality of *SoundScratch*.

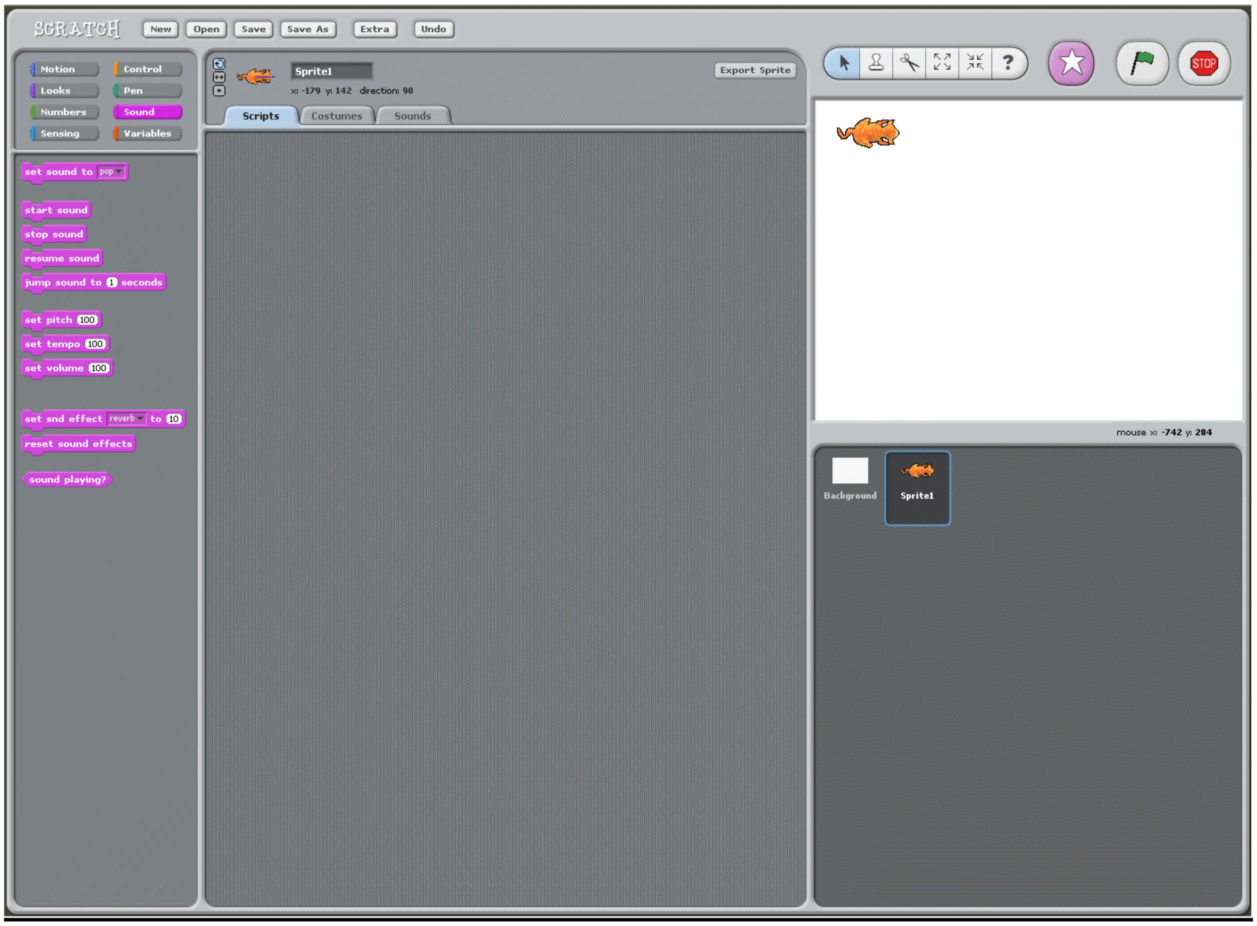


Figure 11: Scratch Startup Screen

Users choose from one of the 8 categories of interlocking blocks. The categories can be seen in the upper-left-hand side of Figure 11 and the corresponding blocks for the category appear below. Users drag the blocks into the script window in the middle, connect the blocks, then click on them to execute their script. Typically users write their scripts to manipulate costumes such as the cat shown on the right.

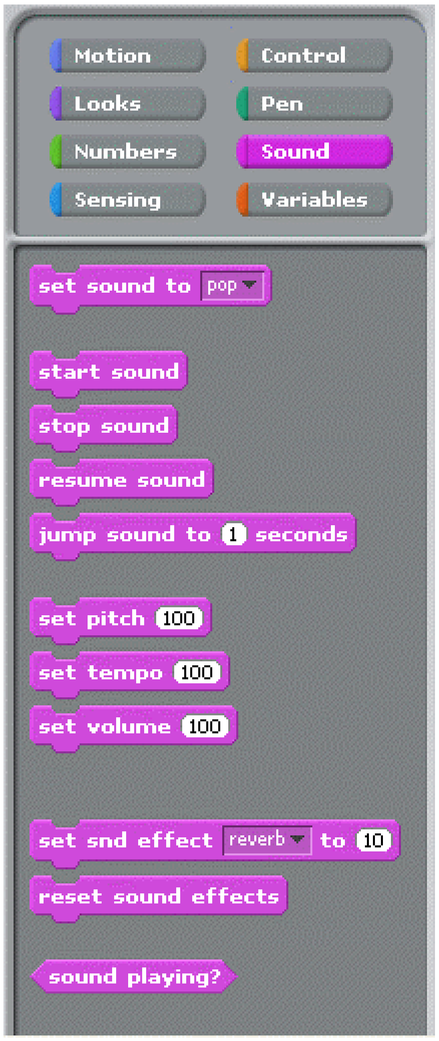


Figure 12: A Close-up of *SoundScratch*

SoundScratch *is a set of blocks that have been added to the* Sound *category of* Scratch. *New projects load with two default sounds, meow and pop.*

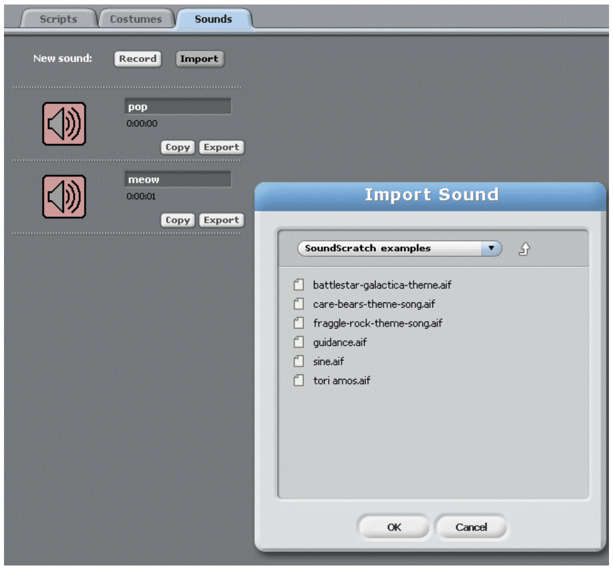


Figure 13: Importing a sound

Sounds can be imported into the environment or recorded on the fly. When a sound is imported into the environment, it will be listed below pop *and* meow *in the* Sounds *category of the script window, as shown in Figure 13. It will also appear in the drop-down menu of the* set sound to *block, which is shown below.*

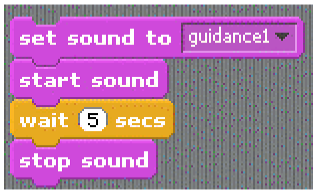
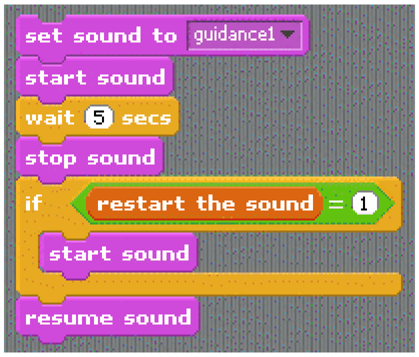


Figure 14: Starting and Stopping

These blocks will play the first 5 seconds of guidance1. Guidance1 *was imported into the project in Figure 13.*

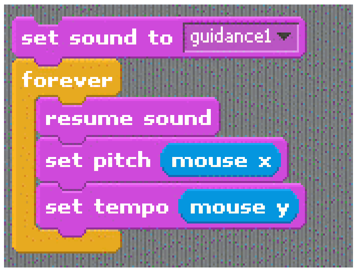


```
set sound to guidance1
start sound
wait 5 secs
stop sound
if restart the sound = 1
    start sound
resume sound
```

Figure 15: Resume

The resume *block starts a sound from where it was stopped. If the sound is already playing, it is ignored.*

These blocks play the first 5 seconds of guidance1. If the variable restart the sound *is set, they will then restart guidance1 from the beginning. Otherwise they will resume playing guidance1 from where it was stopped. In this second case, the sound is essentially uninterrupted.*

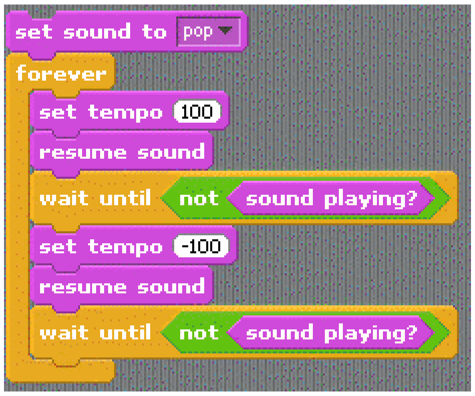


```
set sound to guidance1
forever
    resume sound
    set pitch mouse x
    set tempo mouse y
```

Figure 16: Pitch and Tempo

Set pitch *sets the pitch of a sound independent of speed.* Set tempo *sets the speed of a sound independent of pitch. When speed is a negative value, the sound will be played backwards. If both are set to the same value, the sound is resampled.*

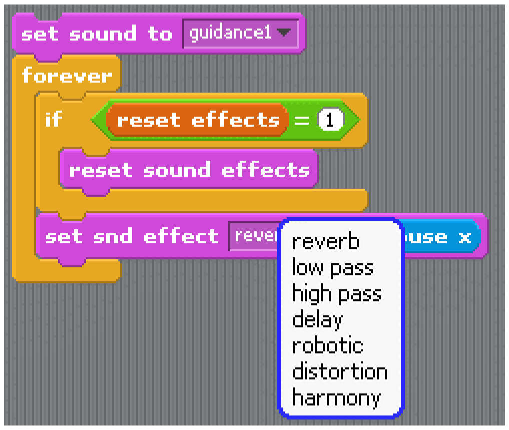
These blocks will continually repeat guidance1 *at the pitch of the current mouse x position and the speed of the current mouse y position. The forever loop is used so pitch and speed are continually updated with mouse movements.*



```
set sound to pop
forever
    set tempo 100
    resume sound
    wait until not sound playing?
    set tempo -100
    resume sound
    wait until not sound playing?
```

Figure 17: Bouncing a Sound

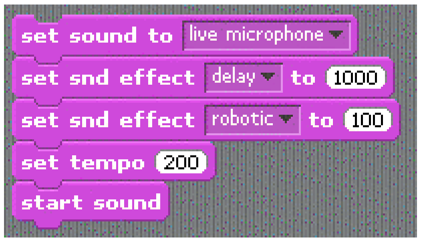
These blocks will play pop *forwards, then backwards, over and over again*



```
set sound to guidance1
forever
    if reset effects = 1
        reset sound effects
    set snd effect rever    use x
```

Figure 18: Sound Effects

Besides tempo, pitch and volume, users can choose between a variety of sound effects, each of which can be controlled in real-time.

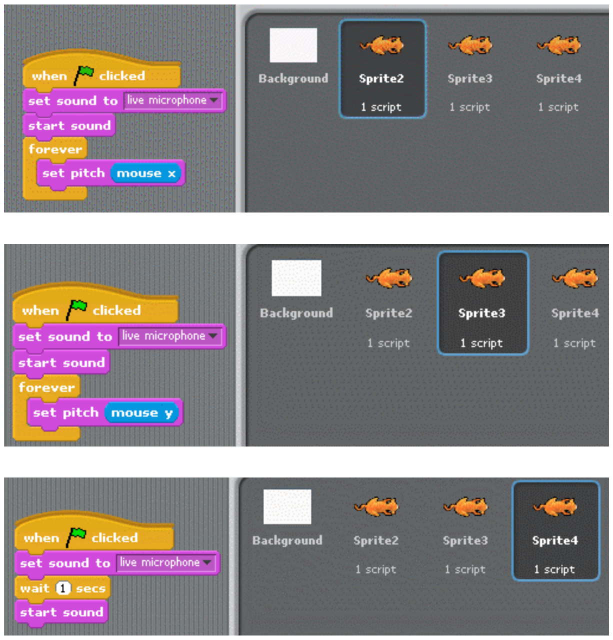


```
set sound to live microphone
set snd effect delay to 1000
set snd effect robotic to 100
set tempo 200
start sound
```

Figure 19: Live Microphone Input

All sound manipulations and effects can also be applied to live microphone input.

These blocks will delay the microphone input 1 second, use a vocoder to create a robotic effect, then play the result back at twice the speed.



```
when [flag] clicked
set sound to live microphone
start sound
forever
    set pitch mouse x
```

```
when [flag] clicked
set sound to live microphone
start sound
forever
    set pitch mouse y
```

```
when [flag] clicked
set sound to live microphone
wait 1 secs
start sound
```

Figure 20: Sprite Independence

Each sprite in SoundScratch *can contain its own independent set of sound manipulations. This set of 3 sprites each takes microphone input. One modifies the pitch of the signal dependent on the position of mouse x, one modifies the signal dependent on the position of mouse y, and one delays the signal one second. All 3 run concurrently when the green flag is clicked. The green flag can be seen in the upper-right-hand corner of Figure 11.*

.

3. Motivation and Historical Perspective

This section reviews the development of learning, electronic music, tangibility, and dynamic systems programming as it relates to *SoundBlocks* and *SoundScratch*. It shows how these ideas are connected, and how they might powerfully interact together, thus motivating the development of the two environments.

3.1 Learning by Design

> "The child is curious. He wants to make sense out of things, find out how things work, gain competence and control over himself and his environment, and do what he can see other people doing. He is open, perceptive, and experimental. He does not merely observe the world around him. He does not shut himself off from the strange, complicated world around him, but tastes it, touches it, hefts it, bends it, breaks it (Holt 1967)."

John Holt believed children learn best when playing and exploring. His idea stemmed naturally from a lineage of educators who had been studying how children learn. Froebel, who originated the kindergarten system in the 1830s, believed that children learn best through activity and exploration of their own environment (Brosterman 1997). More recently, educators have added that this activity and exploration can be focused when the child is the designer and the creator within the learning environment. Specifically, research nows suggest that children:

- Learn best when given tools they find engaging (Papert 1993)
- Develop the deepest understandings of a topic when they are free to create and design within that topic (Resnick, Bruckman et al. 1996)
- Learn important skills about creative problem solving and about their own thinking when correcting mistakes in their own designs (diSessa 2000)
- Are most engaged in learning when inventing things they care about (Resnick, Rusk et al. 1998)

Perhaps children learn best when they are free to love what they create and free to create what they love. Seymour Papert dubbed a term for the philosophy, Constructionism, which he named after Piaget's theory of Constructivism. Constructivism states that children learn best when they are active builders of knowledge. Constructionism states that children are best at being active builders of knowledge when they are building things. In the words of Papert:

> "Constructionism is built on the assumption that children will do best by finding ('fishing') for themselves the specific knowledge they need (Papert 1993) pg. 139."

Papert's work in the field has spawned a new field of educators called Constructionists. Generally, this group might suggest that a good educational tool should support an environment that provides the child with the opportunities to explore, create, and design in a way that is personally meaningful for them.

3.2 Computers in the Learning Environment

Many of the ideas that form Constructionism are, like John Holt's ideas, a natural evolution of many generations of the philosophy of learning. What makes Constructionism such a dynamic philosophy now is that recent technology, specifically digital computing, is the perfect platform to support it. Specifically, programming itself gives children a way to explore by design, to create, and integrate their artistic and mathematic endeavors.

Initial research exploring Constructionist philosophy within technology has centered around LOGO, a computer language which incorporates geometry, functional programming, and mathematics (Papert 1980). Since then, the ideas that LOGO introduced have been extended. StarLogo (Resnick 1996) lets children model complex, emergent, decentralized systems. NetLogo (Tisue and Wilensky 2004) gives children a modeling environment to design and explore complex and natural phenomena. Many other derivations exist. These derivations, like the original LOGO language on which they are based, are primarily visually oriented. They encourage children to use pictures first, then perhaps audio second, as their basis for artistic expression in these languages. The sound capabilities in these systems are very limited.

3.3 The Arts and Learning

Extensive research has been done documenting that children benefit from learning and creating in the arts. Moreover, the arts connect with other disciplines and many feel that interdisciplinary learning richens and deepens the learning experience. The arts are free of learned formalisms. At the same time, artistic creations touch our immediate experiences. We can act spontaneously to our artistic creations and at the same time not know what to do (Bamberger 1979). This is at once a freeing experience and a profoundly educational one. As John Holt explains:

> "I said in **How Children Fail** that the test of intelligence was not how much we know how to do, but how we behave when we don't know what to do. Similarly, any situation, any activity, that puts before us real problems that we have to solve ourselves, problems for which there are no answers in any book, sharpens our intelligence. The arts, like the crafts and the skilled trades, are full of such problems, which is why our skilled artists, artisans, and craftsmen are very likely to be sharp-witted people. Their minds are active and inventive; they have to be (Holt 1967)."

Taken altogether, this makes a compelling case for educational tools connecting technology, learning by design, and learning in all of the arts. This is consistent and complementary with Constructionism. However, as with LOGO, most exploration that connects technology, learning by design and the arts has focused on visual arts. Less research has been done as to how these ideas might connect with music and sound.

3.4 Music and Learning

Within the arts, music offers opportunities and benefits similar to visual arts that can also be used to support the Constructionist framework. It is powerfully expressive, educational, and integrates well with other disciplines. The National Standards for Arts Education states that "music is a basic expression of human culture." (The National Association for Music Education 1994)

Within the framework of Constructionism, some initial work in music has been done. In 1979 Jeanne Bamberger used LOGO's simple sound primitives to develop two mini-worlds:

TUNEBLOCKS I and TUNEBLOCKS II (Bamberger 1979). Both TUNEBLOCKS I and TUNEBLOCKS II were based on Bamberger's ideas that music is best explored through high-level structures such as large note groups. These groups form a mini-world in which users can explore, discover and learn. TUNEBLOCKS, like the LOGO language it is built on top of, supports only one voice at a time. It has no polyphonic capabilities.

Aside from Bamberger's work and some tangible interfaces, however, very little is actually available for music and sound that supports the Constructionist framework. What little is available is mostly focused on high-level structures such as notes and groups of notes. Designers of the environment created these notes, not the child. Typically the child cannot even modify them. As Constructionists generally wish to support different types of learners with different styles of learning, there is ample motivation for more research in sound and music that supports Constructionist framework.

Constructionist philosophy suggests that children will find projects meaningful that connect with their environment and their surroundings. In the domain of music and sound, this could arguably be their own voices and the sounds around them. It may also include the music they listen to, which is to a large extent created and modified with both traditional sounds and sophisticated digital processing. How can we create an environment for manipulating not notes but sound itself which supports the Constructionist framework?

3.5 Analog Synthesizers

3.5.1 The RCA Synthesizer

Initially, electronic sound synthesis consisted of combining mathematical formulae to generate waves. Initial forays used strictly analog components. The first commercial synthesizer, the RCA Synthesizer, was introduced in 1956. Harry Olsen and Hebert Belar, employees of RCA's Princeton Laboratories, designed the synthesizer inspired by a 1949 publication titled *A Mathematical Theory of Communication* (Shannon and Weaver 1949). This publication asserted that it would be possible to generate popular music by manipulating high-level structures based on probabilistic models. The publication greatly influenced the two engineers. Specifically, they attempted with their design to make it possible to manipulate both the low-level sounds of notes themselves and the high-level groupings of these notes.

Although Olsen and Belar were disappointed that their synthesizer did not achieve all of the fluency and control they had wished, the RCA marketing team apparently disagreed. RCA released a 4-disc box set of 45-RPM records titled *The Sounds and Music of the RCA Electronic Music Synthesizer*. At the beginning of the first record, the narrator announces the synthesizer and proclaims it to be "a system capable of producing any sound which has ever been produced and any sound that may be imagined by the human mind (Schultz 2005)."

3.5.2 The Modular Synthesizer

Users programmed the RCA Synthesizer using punched paper and a typewriter-style keyboard. American engineer Robert Moog thought there might be a better a way. He designed the first widely-recognized modular synthesizer: a synthesizer comprised of self-contained connectable units. Moog presented a paper based on this idea at the Audio Engineering Society, which he entitled "Voltage-Controlled Electronic Music Modules" in the fall of 1964. He began accepting orders for his modular synthesizer immediately.

To program a modular synthesizer, users connect patch cords between modules' ins and outs. The resulting program describes the data flow of the synthesizer, and is called a patch. All modules can be connected to each other so any module or any combination of modules can interact with any other combination in any way the user wishes. Because of this, users can explore a large possibility of data flow networks even with only a few modules in a system. Moreover, the modular patch bay system provides an easy interface from which users can experiment to hear the resulting sounds from these networks.

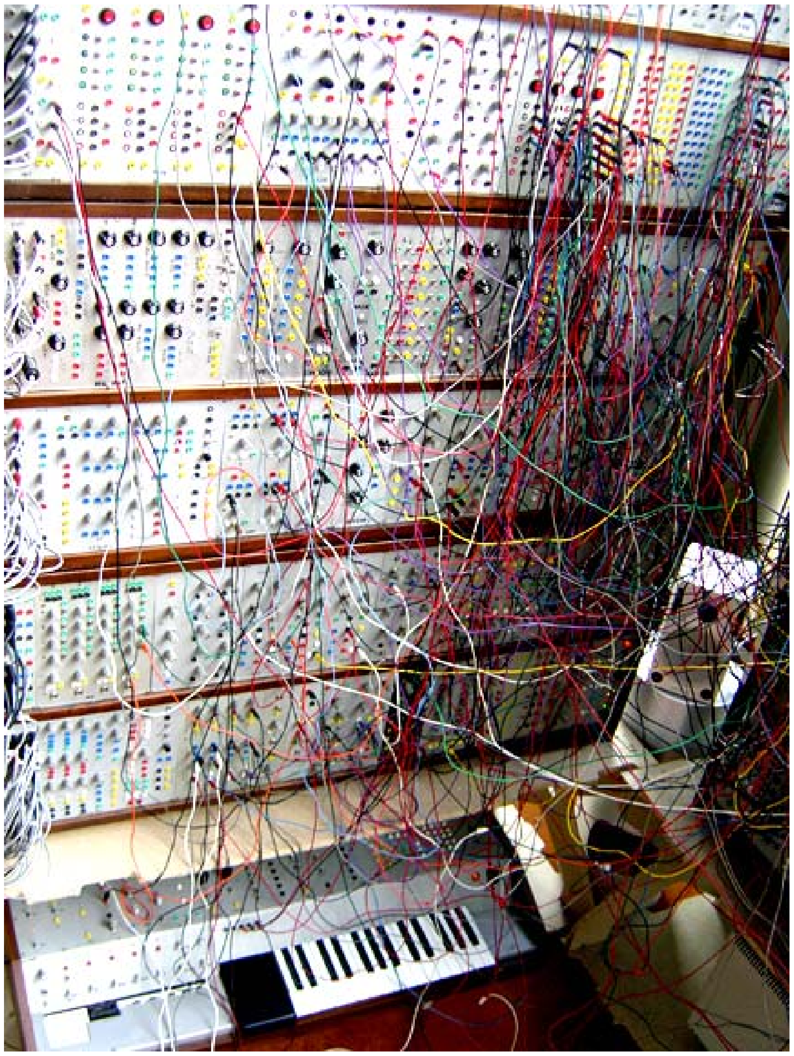


Figure 21: A close-up of cabinets from Joe Paradiso's modular synthesizer

After Moog's pioneering presentation of 1964, the modular synthesizer became an astounding success. In the later half of the 1960s and throughout the 1970s especially, the modular synthesizer was an integral part of a large amount of popular music. Indeed modular synthesizers are still manufactured, and some musicians continue to perform on them as well. Figure 21 shows a close-up taken in September of 2004 of some cabinets from Joe Paradiso's modular synthesizer. Dr. Paradiso performed on the synthesizer at that time as part of *Ars Electronica 2004* (Paradiso 2004).

3.6 Digital Computers in Music: Two Roads

By 1955, people were beginning to explore how digital computers might also contribute to music. That year, computer music pioneer Lejaren Hiller worked with Leonard Isaacson to generate compositional algorithms using a computer. He used these algorithms in his *Illiac Suite for String Quartet,* the first recognized composition with computer-generated material. By achieving this he had succeeded in using electronic means to model high-level structures, exactly what Oslen and Belar had wished to accomplish with their synthesizer. Unlike the synthesizer however, the computer did not also perform the work. It made no sound.

Initially a chemist at DuPont for five years, Hiller had assumed a post as Professor of Music at Indiana University (Hiller and Isaacson 1959). He continued work specifically in algorithmic composition with high-level structures throughout his career. His groundbreaking work in this field created the pathway for the development of algorithmic composition that continues today.

Max Matthews, an engineer at Bell Laboratories and also a pioneer integrating digital computers with music, had an approach quite different from Hiller's (Holmes 2002; Manning 2004). Matthews explored how digital components could be programmed to emulate analog circuits, such as would be found in the RCA Synthesizer. In Matthews' work, algorithmic composition is not the focus. Instead, it is the exploration of the sound itself.

How could sound be designed, created, and manipulated digitally? What does it mean to describe a sound mathematically or programmatically in a computer? Seeking answers to these questions, Matthews and his team wrote the first sound synthesis language, MUSIC I, in 1957. They developed the language iteratively, renaming later versions: MUSIC II, III, IV. With MUSIC IV, they arrived at a powerful computational paradigm to express sound manipulation programmatically. Derivatives of this language include Csound, and are still in use today (Lefford, Scheirer et al. 1999).

Matthews and Hiller paved separate and distinct roads for the digital music community. Hiller's work builds upon our traditional notions of what defines a composer. Before Hiller, a composer chose each note to write down on the page. Hiller showed a way that a composer might instead choose a set of rules or complex algorithmic behaviors that a computer would translate to specific notes. What remains consistent in Hiller's work is the notion that the composer is not a performer and the performer is not a composer. The notes or rules that govern them are chosen before the performance by the composer and will not (intentionally) be changed in the performance outside of the composer's control. In fact, the performance is a completely separate event from composition.

Contrasting this, Matthews' initial work combines our notions of traditional instrument designer and traditional composer. Like a traditional instrument designer, users of Music IV draw from a set of tools by which to describe sound itself, mapping the timbre and envelope of sound to events. Like a traditional composer, users then provide a composition complete with score describing what timbres and envelopes of sound they wished for over time. Contrasting Hiller's algorithmic score descriptions, Matthew's score descriptions were completely defined. The user specified each note's time and duration exactly. The computer, given both the instrument and the score, could then perform the music.

As digital computers continued to advance, Matthews pushed the notion further. Himself a violinist, Matthews explored how simple physical buttons could be mapped to an instrument's controls in real time. This allowed the user, already the instrument designer and composer, to also be a performer within the system. Through this work he eventually developed the Radio Baton, which he continues to refine today (Chadabe 1997).

3.7 Music Concrète

Like Matthews, Pierre Schaeffer was trained as an electronic engineer. However, Schaeffer did not care much for the work of Matthews. Matthews and the other artists and engineers of *elektronische Musik* were synthesizing waves from mathematical functions. Schaeffer wished to manipulate naturally produced sounds. Using variable speed phonographs and tape recorders, Schaeffer recorded sounds, then manipulated them by physically manipulating the turntables and the tape itself. By splicing and joining tape, he could loop and combine sounds.

In 1948 Schaeffer teamed up with composer Pierre Henry, and the two worked closely together to produce the first public performance of *Music Concrète* --- composed manipulation of recorded sounds. On March 18, they performed their *Symphonie* in the École Normale de Musique in Paris. Central to the performance was the performance equipment itself: several sets of turntable, mixers, and loudspeakers. Manipulating the unwieldy turntables in a live situation proved a difficult challenge and the performance went poorly. Nonetheless, *Music Concrète* had begun.

Music Concrète has had a profound impact on the development of popular music. Schaeffer's ideas eventually led to the *Mellotron*, a keyboard that used tape loops and was a precursor to the modern-day digital sampler. Likewise, various common effects units such as flangers, delay and reverberation units initially were controlled by analog tape, inspired by the ideas of Schaeffer and *Music Concrète*. Even the modern-day DJ owes something to Schaeffer, who had been controlling speeds of turntables in the 1930s.

Popular artists of the 60s and 70s especially made extensive use of tape loops. Steely Dan, for example, created the first widely recognized drum loop in the album *Gaucho*. Pink Floyd relied heavily on tape loops for their album *Ummagumma,* and later in *Dark Side of the Moon*. John Lennon and Yoko Ono also did quite extensive experimentation with many of the ideas introduced as *Music Concrète*. It was not easy to make these loops. John Lennon explained how *Revolution #9* was mixed:

> "It has the basic rhythm of the original 'Revolution' going on with some twenty loops we put on, things from the archives of EMI. We were cutting up classical music and making different size loops, and then I got an engineer tape on which some test engineer was saying, 'Number nine, number nine, number nine.' All those different bits of sound and noises are all compiled. There were about ten machines with people holding pencils on the loops - some only inches long and some a yard long. I fed them all in and mixed them live (Miles 1997) pg. 484."

3.8 Digital Sound Manipulation Today

3.8.1 Sound Synthesis Languages

Less than ten years after Max Matthews developed *Music IV*, Barry Vercoe developed *Music 360*, then *Music 11*. By 1985, Vercoe released *Csound*, innovative in part because it was written in C and could therefore be compiled on a variety of platforms. By 1990, computers were powerful enough that *Csound* could create interesting analysis and synthesis in real time (Vercoe and Ellis 1990). More recently, there has been an increasing number of real-time sound synthesis languages. Some, like *RTCmix* and *SuperCollider,* offer a traditional text-based structure. More recent languages, like *Max/MSP*, *Pure Data*, and *VVVV* describe relationships graphically.

These languages, like Music IV that inspired them, are focused first and foremost on the manipulation of mathematically derived synthetic sounds. This is in contrast to our Constructionist philosophy and to *Music Concrète*. We are primarily concerned with manipulating the sounds of the world around us.

This is not to say that these sound synthesis languages cannot manipulate natural sounds. On the contrary, these languages are extremely powerful and can manipulate them in countless ways. In fact, all of these languages offer a universe of possibilities for natural sound manipulation that have yet to be discovered. However, the structures and interfaces of these languages require users to first become familiar with synthetic manipulations such as sine waves, since expressing manipulation of real-world sounds in these languages is more difficult.

3.8.2 Digital Samplers and Commercial Software

Synthesizers have grown up considerably since the days of the RCA synthesizer. Interesting for our purposes, they now encompass most of the ideas and resulting effects generated from the *Music Concrète* movement. Most notably, the modern synthesizer records sounds, which can then be manipulated in the environment in any way we wish.

Consistent with their history, modular synthesizers are manipulated or, in some sense, programmed, by manipulating patch cords to inputs and outputs of various functions. Recently, modular software synthesizers have been developed. These products, such as *ReakTor*, *Reason,* and *VirtualDJ(VirtualDJ 2001-2005)* emulate hardware modular synthesizers. This offers the experienced audio engineer a way to expand his or her possibilities for the manipulation of sound at a greatly reduced price. However, it is highly specialized and is not a true programming environment. Furthermore, it offers little insight for the novice that will help him explore operation of the emulated hardware in a way that might give insight to what might be possible sonically. Therefore, it is not appropriate for development within the Constructionist framework.

Many of these software packages also include a package of presets that produce spectacular "gee-whiz" sounds. Because of this, people with literally no understanding of sound manipulation at all get seemingly impressive results with little or no effort. While this might help software sales, it leaves many disillusioned. They have fun with the presets, but do not find themselves in an environment that they can successfully explore. They may feel bored, uninspired and unsure of what or how to develop further abilities with digital sound manipulation.

3.8.3 High-level music structures

Algorithmic composition continues to be actively explored (Trevino-Rodriguez and Morales-Bueno 2001; Assayah and Dubnov 2004; Adan 2005). Music notation software has also developed significantly (Sibelius 1998-2005; Finale 2003-2005).

Sequences (Cakewalk 1987-2005) and loops in particular have become a central part of popular music. They can now be easily manipulated with such commercial software as *Fruity Loops* (FruityLoops 2003-2005) and *Acid* (Sony Media Software 2005). These products also come with a great package of samples and easy ways to create the common loops we hear in popular music. They give users tools intended to help them imitate popular music, and they appeal in that they are feature-rich and it is easy to get initially impressive results. They are not, however, an environment in which to explore the nature of the sounds themselves.

Developed at the Media Lab, Hyperscore (Farbood and Jennings 2004) offers a creative and playful way for children to draw musical ideas. It is in some sense an environment in which users with no musical training can sketch their pieces visually. The system maps shapes and colors intuitively, producing a MIDI output which people can listen to or have professional musicians perform. Hyperscore is an excellent product, and many users have enjoyed exploring the world of composition with this innovative tool. However, as it is for manipulation of higher-level structures, it does not facilitate nor does its structure lend itself to manipulation of real-world sounds.

3.9 The Scratch Programming Language

Scratch (Maloney, Burd et al. 2004) is a computer programming language for children that is in active development. The language was inspired in part from the observations of Mitchel Resnick and others at Computer Clubhouses (Resnick, Kafai et al. 2003). The youth at clubhouses showed a natural interest in computers but showed less interest in programming. They wanted to use the computer to draw pictures, make interactive art and stories, and play games. Programming languages, even those designed within the Constructionist framework, did not seem to support a direct enough connection with these activities to capture the children's imaginations. Instead the youth were drawn to software where they could quickly draw and create, such as Adobe Photoshop (Adobe Inc. 1990-2005). The Computer Clubhouses are, in some sense, a Photoshop Culture.

Resnick and his colleagues proposed that a computer programming language, *Scratch*, be created which offers youth the power, usability and a direct link to their desire for self-expression that Photoshop offered. Two years later, *Scratch* now offers ways for youth to design and create animated stories, interactive art, and games through programming.

Scratch is an ideal environment within which to explore possibilities for sound manipulation for youth. Experienced Constructionists have designed it from the ground up. It has an appealing interface for children. Already it encompasses a look and feel intended for media manipulation. The program itself is expertly constructed, so extending its capabilities within its framework is simple and straightforward. In short, it is the culmination of years of research connecting youth, education, and technology.

3.10 Designing a Musical Instrument

While *Scratch* offers a wonderful environment within a computer to explore sound manipulation, the computer itself is a potentially limiting constraint. Sound itself is not an expressive device. It is how a sound changes that gives it expressive power (Meyer 1956). The traditional mainstream computer interfaces of keyboard, mouse and screen have not been designed for nor do they provide a transparent way to achieve this capability. Some musicians attempt to perform with traditional computer interfaces, and these performances are sometimes called laptop concerts. However, there is not a way through traditional interfaces to describe physical gesture, a sense of touch or a sense of feel in a natural or familiar way. These qualities are a central part of personal expression. Perhaps for both cultural and physical reasons, they make up to a large extent what it means to play a musical instrument, to see an instrument played, and to emote with that instrument.

Further, tangible interfaces offer possibilities for interactive play and 3-dimensional construction. Within this environment, there are possibilities for networking and dataflow descriptions that may be more intuitive for users if they can see, touch, grasp and feel the manipulations themselves. Future large-scale projects could include multiple sensors, speakers, and microphones, all of which would require development outside the computer.

3.11 Tangible Environments for Sound

In 1997 Hiroshi Ishii and Brygg Ullmer argued for a vision of *Tangible Bits* (Ishii and Ullmer 1997). With the *metaDESK*, the *transBOARD* and the *ambient*ROOM, they demonstrated the need for interactive surfaces, graspable physical objects that connect to internal computer manipulations, and ambient awareness to connect foreground cues with background cues. Ishii and Ullmer further motivated the development of tangible interfaces by demonstrating in their work that the GUI interface is too restrictive. GUI interfaces cannot embrace the richness of senses and skills people have developed through their interaction with the physical world.

A year later, Gorbet, Orth and Ishii developed a modular design for tangible manipulation of digital information topography. Their system, *Triangles* (Gorbet, Orth et al. 1998), consisted of multiple equilateral physical triangles, each with a micro controller and unique ID tag. A host computer monitored the topological configuration of the triangles and mapped various physical configurations with sound and video. *Triangles* described a flexible networking system, and the designers implemented several software environments for triangles within which children could tell non-linear stories through video and audio, or create and trigger various media clips. Moreover, by designing *Triangles*, Gorbet, Orth and Ishii demonstrated that it was possible to create low-bandwidth, computationally inexpensive modular objects which could describe complex physical and virtual structures.

Offering a way to build musical phrases and structures, *Block Jam* (Newton-Dunn, Najano et al. 2003) consists of a set of physical cubes networked to a computer. Users interact with the physical cubes, as well as with a computer screen. *Block Jam* is similar to *Triangles* in that various combinations of blocks can be combined to describe complicated network relationships. However, it differs in that each cube has an LED and a clickable button. Also there are different types of blocks, such as a play block and a path block. By combining these various blocks, users can create nonlinear sequences and describe high-level structures. The environment provides predefined sounds and various high-level rules for loops.

Inspired in part by Ishii and Ullmer's work, there has been recent research related to tangible interfaces for music systems. The designers of these systems, for practical reasons, often limit the environment to a table top or similarly-sized surface. For example, the *Audiopad* (Patten, Recht et al. 2002) is a composition and performance instrument for electronic music which tracks the positions of objects on a tabletop surface and converts their motion into music. Using RFID tags, the system lets the user pull sounds from a giant set of samples to create melody and rhythm. *Audio d-Touch* (Costanza, Shelley et al. 2003) provides a learning environment for music composition and performance, tracking tangible blocks with a camera mounted above a tabletop. The *Music Table* (Berry, Makino et al. 2003) is a composition system that provides a tactile and visual representation of music which can be manipulated to make musical patterns. *Musical Trinkets* (Paradiso, Pardue et al. 2003) uses RFID technology to map various sounds and musical gestures to up to 30 unique objects, as a user moves them freely within the vicinity of a tag reader. There are many others, including *Jam-o-Drum*, *reacTable**, *Instant City*, *Audiocube*, *Scanjam*, *Smallfish*, *Lemur*, *Yamaha Music Table*, and Fisher-Price's *Play Zone Music Table.*

The music controller has become a common tangible interface for sound. With a music controller, the user can manipulate sensors. Manipulation of the mappings of the sensors is typically done through virtual means with existing software. For example, the *Adaptive Music Controller* (Merrill 2004) is a hardware device with predefined sounds. The device is hand-held, and a user can train the system by mapping motions they make while holding the device to sounds predetermined in the system using Pd. The system uses an Inertial Measurement Unit called a *Stack* (Benbasat and Paradiso 2005) to track motions in all three axes. The *Sonic Banana* (Singer 2003) is a two-foot long flexible rubber tube with a network of four bend sensors and a single pushbutton switch. The sensors send continuous raw data in the form of MIDI controller parameters. Typically, the *Sonic Banana*'s data is then routed to a Max/MSP patch where it is mapped to sounds, much in the same way as the *Adaptive Music Controller.* There are numerous similar examples of this approach, including *BeatBugs* (Aimi 2002), and *Shapers* (Weinberg, Orth et al. 2000).

4. SoundBlocks: Initial Work

Figure 22: Interacting with *SoundBlocks*

4.1 SoundBlocks

Eleven principles have guided the design of *SoundBlocks*:

4.1.1 Inexpensive

Digital sound manipulation should be accessible to everyone. For this to be possible, any digital sound manipulation environment must be commercially viable. It cannot be cost prohibitive. This is an extremely challenging confinement. Lego estimates that a maximum of 10% of the consumer price of an item can be used to address technological implementation cost (Risvig 2005). This means that a $50 product can have only $5 worth of electronics in it.

4.1.2 Does not require a traditional Windows, Linux, or Macintosh computer

There are numerous disadvantages to systems that are dependent on standard Windows, Mac OS, and Linux PCs. Such computers are typically general purpose, meaning that they are used for a multitude of tasks and, therefore, are not necessarily available at convenient times. They often require initial setup, and software maintenance seems continually necessary to maintain smooth operation. Even after installation, computers are often inconvenient, as software needs to be run manually by the user. Traditional computers often have to be booted up and their state cannot be predicted before operation due to the variety of users that might use a particular computer. Creating robust software for generic hardware is a continual challenge for software developers, and glitches are common. Specifically, glitches in sound are notorious with all operating systems, as these systems are not designed with a high priority given to real-time manipulation.

Perhaps equally important to the reasons stated above, computers are not generally viewed by our society as a playful environment for design and exploration. Instead, they are often viewed as either a tool for productivity or a game machine. This could conceivably hinder a person's creative approach when exploring an unfamiliar environment.

A traditional computer also inhibits portability of the system. If the blocks require a traditional computer, for example, a user cannot easily carry the blocks with her to the grocery store, to her friends, to the dining room table, or on the floor. A traditional computer draws high current, so inexpensive battery power can be an insurmountable challenge.

Traditional computers are also relatively expensive. This expense challenges the playful environment *SoundBlocks* might otherwise provide, as parents, teachers, and the children they look after become concerned with proper care of the computer to preserve it as an investment.

4.1.3 Scalable

Although initial prototypes as realized for this thesis may only support a limited number of operations with a few blocks, the infrastructure the system provided should allow development of a nearly limitless number of blocks, with each block capable of supporting a high level of sophistication. The blocks should be capable of supporting arbitrarily complex operations and allow unique descriptions of state and configuration as provided by the user. A future programming environment for users to program the blocks might even be possible. The structure of the system should allow blocks to have a reasonably large number of inputs, outputs and sensors on each block.

4.1.4 Aesthetically pleasing

If the blocks are to be a tool for artistry, they should themselves be artistic. This in itself provides some initial inspiration for users. For example, most direct to my own work, the violin doubles as a tool for musical expression and a visual work of art in its own right. Even a carpenter's finer tools often exhibit an aesthetic sense.

4.1.5 Intuitive to the User

Users should not be forced to learn to describe manipulations that serve to be convenient for computer or mathematical representation. Instead, functional relationships within the environment should describe classes of operations which users find expressive. These classes can and should be "black boxed." It is not important that the user understand the low-level choices for what mathematical operations describe the sounds they are hearing. Instead, the system should be constructed to match as closely as possible the intuitions that non-technical users expect when manipulating the blocks, even if this requires classes of functions and mathematical mappings of these classes between them.

SoundBlocks might be more intuitive for users if the blocks can alter their behavior based on the configuration of the network. Such a context-aware, adaptive property must be designed with care. If blocks are adaptive in an arbitrary way, the environment could appear confusing and unpredictable to the user. On the other hand, some sorts of adaptive behavior hardly would appear adaptive at all to a user. A delay block, for example, should be able to delay any type of data it is given.

4.1.6 Provides Instantaneous Feedback

SoundBlocks should respond to both network changes and sensor feedback instantly. Creative exploration within the domain of sound requires this instant feedback. It would be nearly impossible, for example, to learn to play the piano if the response from the piano were delayed and not consistent. Professional musicians, in fact, are typically uncomfortable with digital sound environments that respond with more than 10ms delay.

4.1.7 Complements audio feedback with visual feedback

Visual feedback provides both an alternative way to understand the network state and a way to capture the description of some properties that can be difficult to describe with audio alone. Providing visual feedback can also support another dimension of understanding to the audio environment during real-time use. For example, VU meters provide visual feedback so a user can easily gauge the amplitude of an audio signal. This type of feedback can be useful for understanding the behavior of an environment and can also aid in debugging it.

4.1.8 Primarily provides children with a means for personal expression through physical interaction

If the environment does not in some way inspire users to create sound in personally meaningful ways, they will quickly become bored by it. If this happens, the cycle of learning through design and learning through play is broken. The environment might, at best, then become an artificial pedagogical tool for understanding digital sound manipulation. Such tools already exist and are not the intention for this environment.

4.1.9 Robust

The environment is intended for those who may not and should not be required to understand anything about digital circuitry or computers. Because of this this, various unpredictable situations can occur. If these situations expose error messages and code to the user or require the system to restart, they can be very distracting from the intention of the environment. As much as possible, error messages should be hidden and code execution should attempt to recover from all error conditions. In addition, the hardware should be tolerant of shorts, brownouts, and loose connections.

4.1.10 A physical construction kit

Some of the inspiration for manipulation of the environment may include its physical construction. Therefore the blocks should be easily manipulated physically to assume various positions with each other. They should be able to twist and turn around each other. If possible, three-dimensional manipulations should be supported.

4.1.11 Offers complementary learning in fields other than sound

Motivation for kids to explore the environment will come from their desire to explore sound and to express themselves. However, the environment will be designed to offer as a byproduct learning in other areas as well. Natural avenues for this learning include programming, understanding data flow and network structures, and mathematics.

4.2 First Iteration: Design, Implementation, and Observations

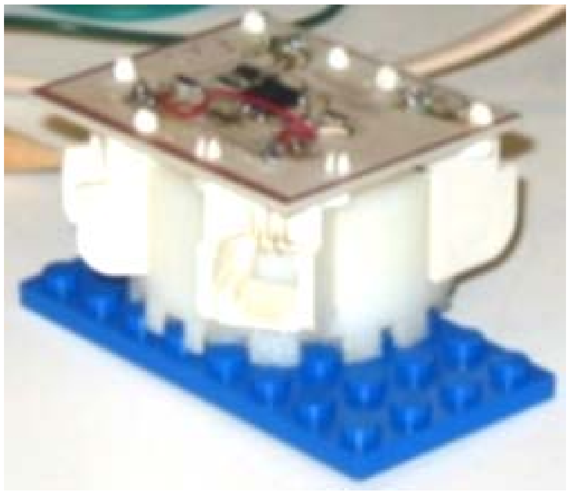


Figure 23: First iteration Interlocking footprint

4.2.1 Design and Implementation

The first iteration of blocks was intended as an experiment of what might be possible and how it might be constructed. It consisted of 6 blocks: 2 pitch shifters, 2 delay blocks, a number block, and a host. Each block was built around an Atmel TINY15L and included a red LED. The blocks used a peer-to-peer communication system based on 1-wire UART, and could transmit data at 19,200 baud. They connected to each other with telephone jacks, telephone plugs and telephone cords. The block housing was a simple footprint which interconnected with standard Lego's. Python code polled the network for network configuration and sensor data. This information was then passed with sockets to Pure Data (Pd), a graphical sound synthesis language (Puckette 1997).

I chose Pd as my sound synthesis language for several reasons. First, Pd is an interpreter not a compiler. Therefore it is a straightforward process to dynamically create Pd code (called patches) on the fly. What this means is that, while Pd is running a patch I can send it commands to extend this patch or write a new patch and the changes will be integrated instantly. This was very useful when updating Pd for network configuration changes. Second, the graphical nature of Pd allowed me to experiment with the idea of mirroring the physical block world with a virtual world on screen. Third, Pd has already been extended to run on PDAs. This meant that, perhaps, Pd might provide an easy way to transition away from needing a conventional computer in future development.

In Pd I constructed blocks that mirrored exactly the functions of the blocks I had built. Each block had, nested within it, a Pd patch. This type of block construction is analogous to class construction in a traditional text-based programming language. When the user looks at the computer screen, she would see the same blocks as those she had networked together in the physical world.

Choosing where to put the blocks on the screen in Pd turned out not to be trivial. I used a springs and masses approach to calculate their position relative to each other. Each block was given a mass and would therefore experience gravitational pull toward its neighboring blocks. At the same time, each block was repelled from its neighbor blocks by imaginary springs. At start time, I would create the virtual world of blocks by placing the blocks randomly. Then I would iterate through time periods, moving the blocks as dictated by the laws of physics. When the system arrived at a steady-state condition, I considered this the final position of the blocks and would send this configuration to Pd.

The 1-wire UART protocol I used had separate lines for power and signal. Therefore it was necessary to have 3 wires connect each block. Telephone cord, plugs and jacks offered the cheapest solution by far for providing these connections. It also made for very quick and simple construction of cables. Using a crimper, bulk telephone wire, and telephone plugs, I could create telephone cables of any size in a matter of seconds.

Block inputs and outputs appeared to the user as either control data or audio data. This is the traditional technique for describing networks of sound synthesis, and is used by Csound, Pd and most other sound synthesis languages. Internally to the blocks, no audio data was actually present, since the blocks serve only as a tangible interface for the central processing host. Instead, audio data is represented within the network as audio-parameter data.

I constructed four types of blocks: a host block, "type A" blocks, sensor blocks, and a number block. A host block converts from standard 2-wire serial communication from a PC to the 1-wire UART that all other blocks in the system use to communicate. "Type A" blocks consisted of two inputs, two outputs and a red LED. Both outputs sent the same data so in a sense it would also be accurate to describe each block as consisting of only one output and a built-in Y splitter for this output. A type A block represented a standard sound synthesis function. The choice to call these blocks type A has no special meaning. Sensor blocks consisted of no inputs, two outputs, a red LED and some sort of sensor. They would send to both of their outputs whatever sensor data they were receiving. A number block consisted of no inputs, two outputs, a knob, two buttons and a 3-digit 7-segment LED.

I implemented pitch shift, delay and a mixer all as type A blocks. The pitch shifter's inputs were an audio-parameter signal that the pitch shift would be applied to, and a control rate describing how much to pitch shift to apply. The delay block's inputs were an audio-parameter signal that the delay would be applied to, and a control rate describing how much delay to apply. The mixer block's inputs were both audio-parameter signals. Its output would be the sum of the two audio-parameter signals.

I created 2 sensor blocks, which were both ultrasonic proximity detectors. These consisted of both a transmitter and a receiver. The transmitter would send a 40kHz signal and the receiver would measure the number of milliseconds it would take to receive the echo of this signal. It would send this number as its sensor data. The sensor blocks had a range between 5cm to 1m.

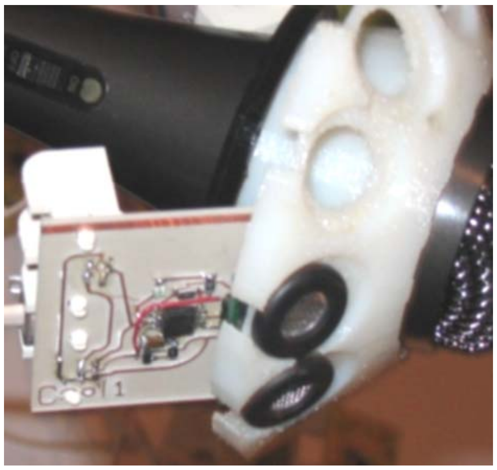


Figure 24: Proximity Block

I also created a number block. This consisted of a potentiometer, 3 seven-segment LED displays, a red LED, and two buttons. The user could dial a value by turning the potentiometer, and then choose the range of this value with one of the two buttons. He could choose whether or not the number was positive or negative with the other button. For example, a user might dial any number between 0 and 999 with the potentiometer. By pressing the range button once, the decimal place on the LED display would shift to show 99.9. Pressing the range button repeatedly would shift the value to 9.99, then .999, and finally return it to 999. Pressing the minus button once would light the red LED that was positioned to the left of the 7-segment LED displays and thus appeared like a negative sign. Therefore the value would then read –999. Pressing the minus button again would invert the value again, returning it to 999. The number block sent the value displayed on its LEDs as its sensor data.

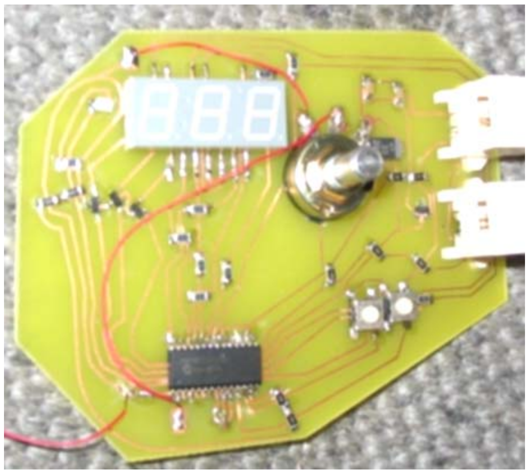


Figure 25: First iteration number block

Communication between blocks was peer-to-peer and included no buffering, even at the byte level. This meant that, as a block received data from a block connected to its input, it would immediately pass this on. A block could directly communicate only to its parent block and its child blocks. Blocks understood a simple set of commands, to allow for realization of the network topology and the ability to receive sensor data. The commands allowed network discovery through depth-first-search, and allowed for circular relationships.

4.2.2 Observations

Several design choices became clear through development of the first iteration:

1. **Flexible cables are problematic and challenge development in 3 dimensions.** The telephone wire used in this first iteration meant that any block could be connected to any other block. There were no limitations on what could be connected to what from proximity or physical location. While this at first might seem like a great and powerful situation, the problem is that it also means that relationships are confusing. Too easily the system appeared as a jumbled mess of wires instead of an organized and clear illustration of network flow. In fact, the 2-dimensional screen representation of the block network structure was often clearer than the network structure itself.

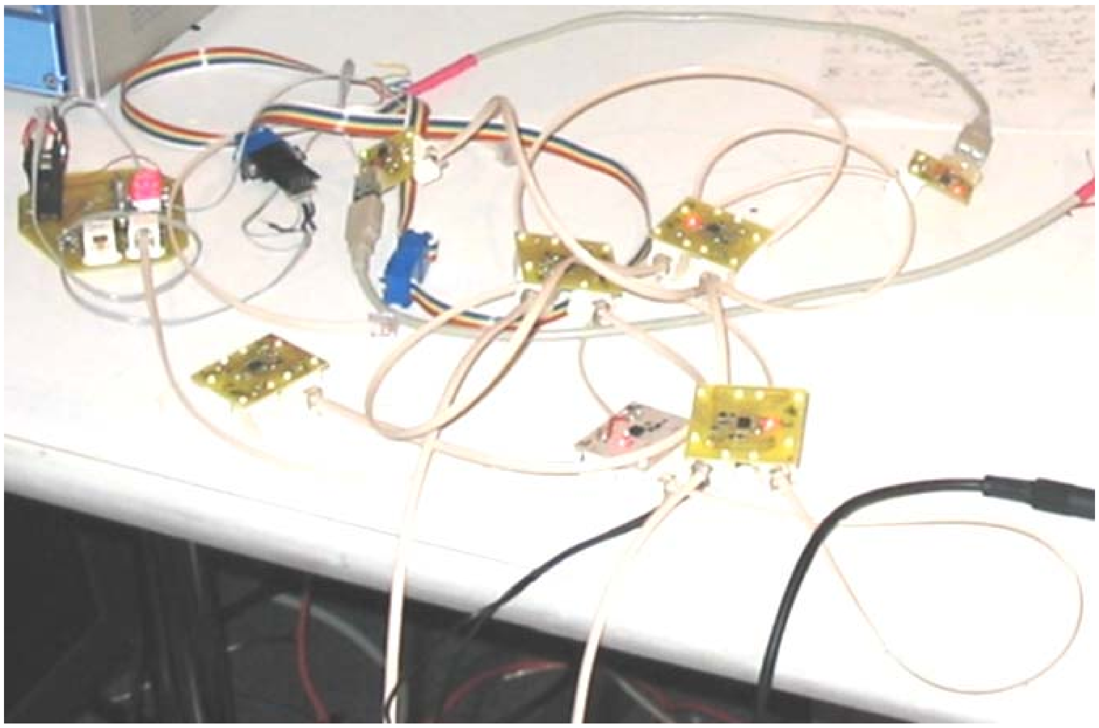


Figure 26: First iteration in testing. Are flexible cables problematic?

Furthermore, flexible cables cannot support any weight so it is impossible to build a 3-dimensional structure with the network itself. It was still possible to create a 3-dimensional structure using Lego's, since the blocks had form factors that fit with Lego. However, this was not intuitive for users as the network had little else to do with Lego in its current state of development.

2. **A visual virtual representation of the blocks distracts from the purpose and exploration of the project.** Many people on exploring this first generation of blocks became very interested in the mapping between the physical representation they created and the virtual representation generated by the computer mapped to their physical representation. This is an interesting situation and has been and will continue to be explored in other projects. However, for the purposes of understanding and manipulating sound it served as a distraction. It put the users' focus too much on the virtual world and what the computer is doing instead of keeping their focus on expressive elements with which to manipulate sound.

3. **1-color LEDs do not provide enough feedback.** The LEDs were too limited in their use for debugging to express meaningful information to the user. At best, they showed when a block was activated. Beyond this, LED mappings proved counterintuitive to users. LED brightness seemed too vague in general to convey meaning, and various lighting situations made this even more of a challenge. Beyond this, LEDs could light to show errors, extreme changes or areas of range, and none of these mappings proved intuitive to users. A user would see the LED do things but have no sense of what the mappings might be.

4. **Peer-to-peer networking offers perhaps unnecessary challenges to implementation.** Although we wish for the user to understand the blocks as having a peer-to-peer relationship, actual implementation of this is a challenging task. Peer-to-peer networks are still a hot topic of research, as traditional solutions offer challenges in robustness and can be time consuming to implement. Our own peer-to-peer network suffered from continually corrupted data and proved to be very difficult to debug. If there were an alternative way to convey the same sort of information that the blocks were conveying in their peer-to-peer network, it would be good to examine this.

5. **Requiring more than two-wires between the blocks is limiting.** Because more than two wires were necessary to send power and data to and from the blocks, the types of connectors necessary to connect blocks together limited the system. For example, connectors could not rotate around each other, which would expand the possible physical relationships between the blocks. Further, less traditional connectors such as magnets would pose challenges with alignment.

6. **Form factor is important.** The blocks offered no personality in terms of their shape and size. Many users commented on this, and suggested that the blocks might feel more playful if they had a more appealing shape and the inner circuitry were more hidden.

7. **A more adaptive system would offer a richer experience, discovering understanding through exploration.** This first generation offered no significant adaptive properties. Only users who already had experience with digital sound manipulation found the environment playful. Other users were forced to try to

understand digital sound manipulation first before they could get appealing results from it. Furthermore, the system did not offer a rich environment. Only the handful of networks I had considered when designing the system were interesting. No significant new relationships could be discovered. If the blocks had more adaptive properties, a richer environment might be possible.

Most apparent, users had trouble distinguishing between audio and control values. The distinction, necessary to create sound within the environment, served as an unnecessary distraction which was convenient only for the computer and its functions.

8. **Numbers are not necessary for digital manipulation of sound.** While the number block was fun and interesting, it became clear from its interaction with the network that the actual numbers it was sending were not helpful in the user's experience for manipulating sound. Instead, just as was discovered with the visual virtual representation, numbers in the LED display actually distracted the user from focusing on what these numbers were actually doing --- changing the audio. What the user needed to be aware of was what range of values they were expressing within the system. The actual number this represented was unimportant.

   Numbers also put too much focus on the mathematical operations of the environment. It might create a fun environment for future engineers, but our intention is to support multiple types of learning. We wish to share focus with the expressive elements.

9. **User parameters should be as limited as possible while still allowing for a broad range of expression.** It is powerful and inspiring to consider that users might create whatever they wish, limited only by their own imaginations. What can be overlooked in this lofty goal is the observation that constraints within a system help to focus users, and that infinite possibilities for expression may exist within these constraints. Traditional instruments demonstrate this clearly. For example, a violin has a very narrow range for dynamics and tonal shapes. In spite of these constraints, it is considered an extremely expressive instrument, and its boundaries for personal expression continue to expand today, more than 300 years after its invention.

Considering this point of view, it became clear that blocks should be limited to as few user parameters as possible while still allowing for great personal expression. Other parameters should be assumed and/or should change with the user parameters as mapped within the design structure of the system. How and what the user parameters would map to and how they might be mapped to other internal parameters is a crucial component to making the system easy to use, intuitive and powerful for users.

10. **Values should be mapped to ranges intuitive to our ear, instead of as might be traditionally used in the digital audio field.** For example, frequencies are typically measured in cycles per second, or Hz. Using this scale, the lowest sound we might hear is 20Hz and the highest (for young people) is 20kHz. With this scale, to jump an octave requires a jump of only 20Hz at the lowest end of the audio range and a jump of 10,000Hz at the highest end of the audio range. This logarithmic scale is not intuitive to users. Moreover, building a system where a sensor could express a given range of audio values would require the user to perform various conversions using mathematical blocks. Instead, the system should already internally use mathematical mappings to perform these conversions for us.

11. **Values in the network should be internally smoothed.** Although a user might create networks that quickly shift parameters between various values, the discontinuities in the audio, often referred to as "zipper noise," are unlikely to be a desired audio effect. Therefore, all blocks should smooth out value changes to reduce or eliminate this.

12. **Inputs and Outputs must be distinguished.** Conventional modular synthesizers have the same jacks for their inputs and outputs. This means that two outputs or two inputs could be connected to each other. There are some advanced situations where this is a powerful concept, and professional users have at times made use of this flexibility. Therefore, the first generation of *SoundBlocks* supported this paradigm; all of its jacks were the same and all of the connectors the same on both ends. Unfortunately, users often became confused as to what jacks were inputs and what jacks were outputs. This lack of a distinction made using the blocks more difficult to explore freely and more frustrating, as many connections that physically worked resulted in no sound. If input

jacks were purposely different from output jacks, a user would be physically prevented from many nonsensical connections.

5. SoundBlocks: Final Design

Based on the eleven principles outlined in section 4.1 and the lessons learned from the first iteration of *SoundBlocks*, I developed the *SoundBlocks* environment. Much of the low-level work, specifically design and implementation of the protocol and physical construction of the blocks, was done in partnership with Andrew McPherson. Andrew submitted a thesis on this work (McPherson 2005).

The first half of this chapter looks at the basic structure of *SoundBlocks*. Section 5.1 examines how users perceive *SoundBlocks*. How do they experience the environment? What do they see? What do they hear? Section 5.2 looks at the internal communication between the blocks and the computer, necessary for the centralized internal processing structure of the system. Section 5.3 explains the basic structure of each block and section 5.4 gives a quick overview of the network protocol between the blocks. Since the aesthetics of the blocks are important, we look at block housing in section 5.5. We discuss the functions and various modes of the LEDs in each block in section 5.6. In section 5.7 we examine the structure of the sound synthesis code and how this code communicates to other modules within the computer environment.

We begin the second half of the chapter by examining context-aware behavior of the blocks in section 5.8. Section 5.9 defines the exact behavior of each block. Finally, we conclude in section 5.10 with an analysis rationalizing the decisions specific to each area of implementation.

5.1 The User's Perspective of SoundBlocks

5.1.1 Interacting with the blocks

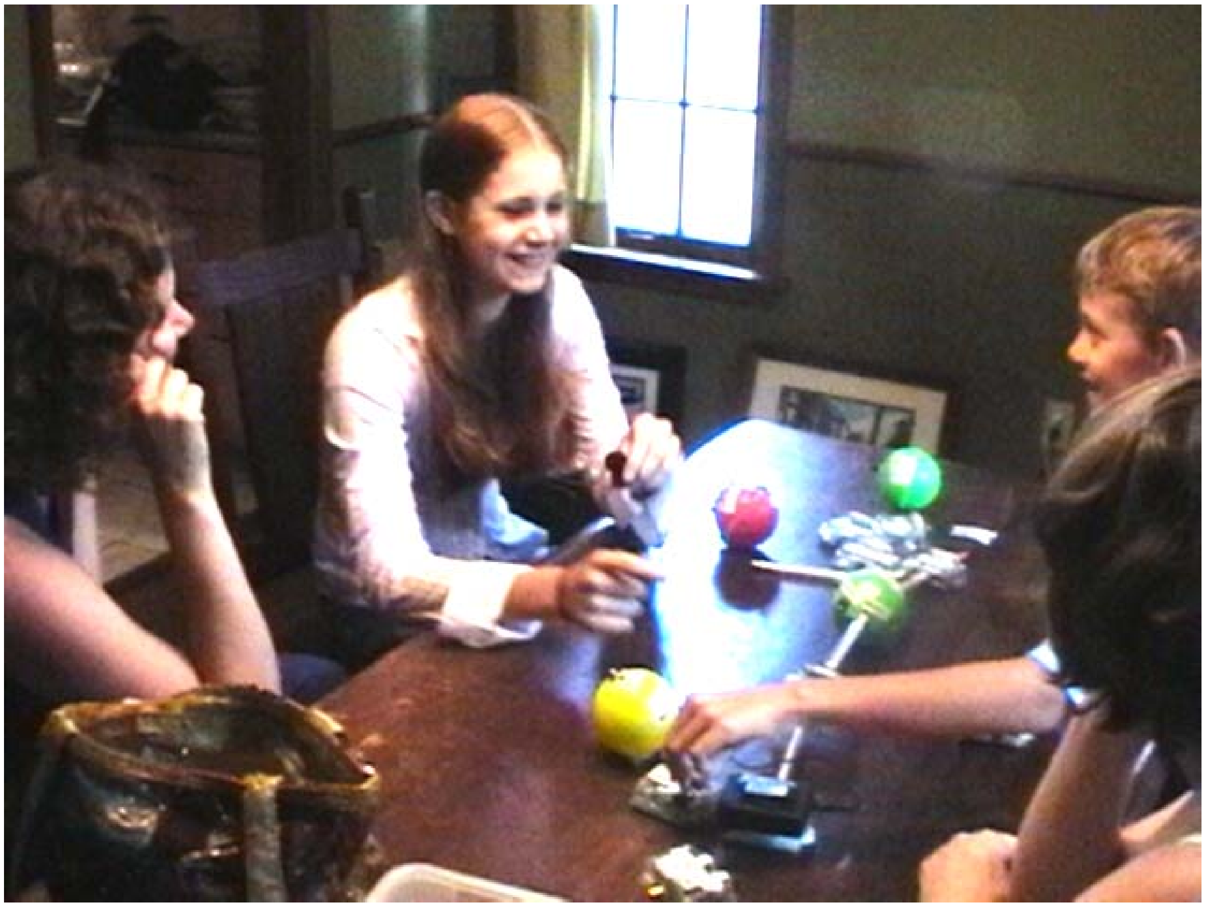


Figure 27: Playing with SoundBlocks

From the user's perspective, *SoundBlocks* is currently a set of 14 blocks, each of which can generate, manipulate or store sound. Each block has a name intended to appeal to kids and also to give some intuition as to what the block does. It also has one output jack, between 0 and 4 input jacks, and an RGB LED, which can show various shades of light. Some blocks also have sensors: buttons, a knob, or a microphone.

SoundBlocks also includes a computer, speaker, and host circuit, which are placed away and even hidden from the user. Ideally the user will ignore these and remain focused on the blocks themselves. A long RCA cable runs from the host circuit to the location of the blocks.

To form a network, the user begins by attaching a block to the cable connected to the host circuit. She then connects blocks to this parent block by selecting among the 9 semi-rigid cables of various lengths. She can continue this process, connecting blocks to each other in any way she wishes. The resulting network describes a set of sound manipulations.

Besides manipulating the blocks, a user can also interact with the system using the sensors. As the user connects and disconnects blocks, turns knobs, presses buttons or speaks into the microphone, he hears the results immediately. At all times the network creates sound as determined by the configuration and state of the network and sensors.

The RGB LEDs show both the state of the block and the flow of the signal in the network. When the network is in its default behavior, each block's LED lights one at a time to show the sequence of the network flow. The color of the LED as it lights shows the state of its block. When a block's state changes suddenly and in a potentially significant way, the sequence is interrupted so this block's LED can light and thus show the block's new current state.

5.1.2 The user's experience with the blocks: as instrument, programming language, and toy

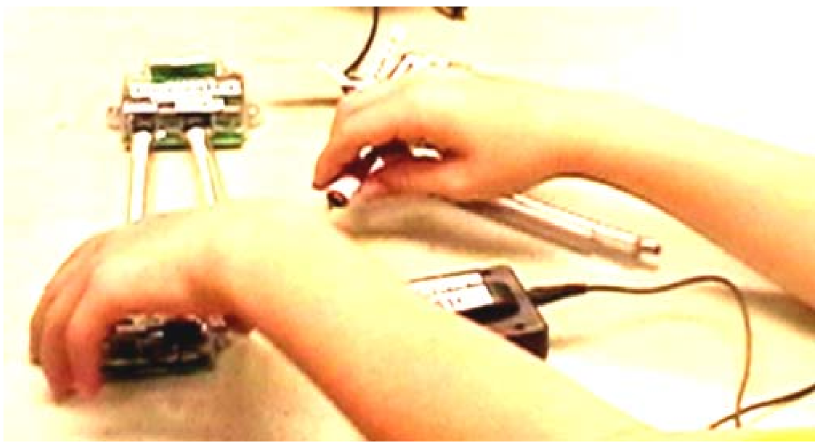


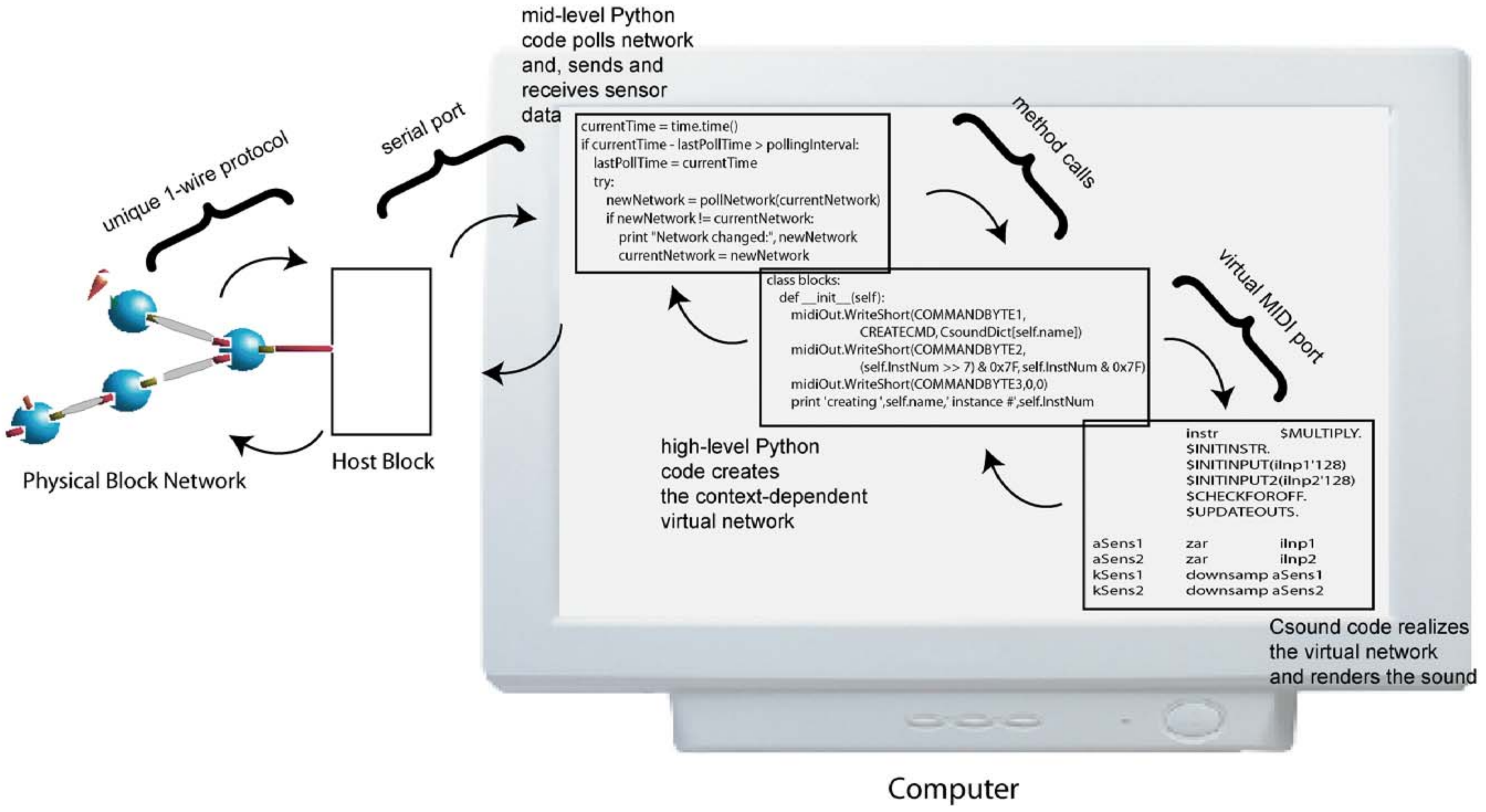
Figure 28: Manipulating *SoundBlocks*

Users may experience *SoundBlocks* in many different ways. In one sense, *SoundBlocks* is a musical instrument. The user designs the instrument by creating the network configuration, then performs on it by interacting with the sensors.

However, *SoundBlocks* is also a tangible programming language for sound. The user writes his code by creating the network topology. He runs this code, interacts with it using the sensors, and observes its behavior by listening to the resulting sound and watching LEDs. If the behavior is not what the user expects, he can debug the system with the feedback he has seen and heard.

And *SoundBlocks* is also a toy. Users are amused to hear their voice, the voices of their friends, and the sounds around them changed in strange and exciting ways. It is easy to create and change sound with *SoundBlocks*, and the manipulations lend themselves to interactive play and games.

5.2 Signal Flow within SoundBlocks


unique 1-wire protocol
serial port
mid-level Python code polls network and, sends and receives sensor data
method calls
virtual MIDI port
Physical Block Network
Host Block
high-level Python code creates the context-dependent virtual network
Csound code realizes the virtual network and renders the sound
Computer

```
currentTime = time.time()
if currentTime - lastPollTime > pollingInterval:
   lastPollTime = currentTime
   try:
      newNetwork = pollNetwork(currentNetwork)
      if newNetwork != currentNetwork:
         print "Network changed:", newNetwork
         currentNetwork = newNetwork
```

```
class blocks:
   def __init__(self):
      midiOut.WriteShort(COMMANDBYTE1,
                   CREATECMD, CsoundDict[self.name])
      midiOut.WriteShort(COMMANDBYTE2,
                   (self.InstNum >> 7) & 0x7F, self.InstNum & 0x7F)
      midiOut.WriteShort(COMMANDBYTE3,0,0)
      print 'creating ',self.name,' instance #',self.InstNum
```

```
         instr          $MULTIPLY.
         $INITINSTR.
         $INITINPUT(iInp1'128)
         $INITINPUT2(iInp2'128)
         $CHECKFOROFF.
         $UPDATEOUTS.

aSens1   zar            iInp1
aSens2   zar            iInp2
kSens1   downsamp aSens1
kSens2   downsamp aSens2
```

Figure 29: Signal flow internal to the *SoundBlocks* Environment

To the user, the blocks themselves appear to manipulate the sound itself. In reality, the blocks serve only as an interface. A computer discovers the physical network of the blocks, translates this physical network to a virtual one using context-aware logic, communicates sensor data from the physical network to the virtual one, and relays LED values or other relevant data from the virtual network to the physical one. This architecture is implemented as a 5-layer hierarchy, and is illustrated in Figure 29. An overview of the hierarchy is as follows:

1. The blocks themselves are the lowest level of the architecture. They contain assembler code used to send values to a host circuit, receive values from this same host circuit, control internal state, and aid in network discovery.

2. At the next level, a host circuit connects the blocks to the computer's serial port. It translates the blocks' 1-wire power-and-signal protocol to standard UART. (Consistent with terminology within the field, we call the protocol 1-wire since power and signal are both on one wire. Technically this is a 2-wire protocol, since a ground wire is also needed.)

3. Mid-level Python code in a computer polls the network through the serial port. It sends changes in network configuration and sensor values from the blocks, to high-level Python code through method calls. It also receives LED values from the high-level Python code and passes it to the blocks. All communication is initiated by the mid-level Python code, which runs in a continuous loop. This loop polls the blocks for network changes and sensor values, and polls Csound via high-level Python code for LED values.

4. High-level Python code uses context-aware logic to create and/or modify the virtual network. It sends this updated network description through a virtual MIDI port to Csound. It also receives information through a virtual MIDI port from Csound regarding the current state of the virtual network

5. At the highest level, Csound, the sound processing language, creates or modifies its virtual network and its current state as instructed. It then renders the resulting sound.

All communication within the hierarchy is initiated by the mid-level Python code. It runs in a continuous loop, polling the network, reading sensor values, and updating LED states. Here's a typical scenario describing how the 5 modules work together.

1. *SoundBlocks is started and no blocks are connected to the network.*

    a. The mid-level Python module begins its continuous loop. The loop polls the network via the host circuit looking for changes to the network every ¼ second.

2. A user connects Micky Mic *to the host circuit, as shown in 2.1, Figure 3.*

    a. Within ¼ second, the mid-level Python code detects the change. It then polls network, and discovers *Micky Mic.* It assigns *Micky* a polling number and reads *Micky's* block type, serial number, the number of inputs *Micky* has and the number of dimensions of sensor data *Micky* reports. It calls the high-level Python code with this information.

    b. The high-level Python code determines what *Micky Mic*'s correct function and state should be, given the network configuration. It instructs Csound to create an instance of this function by sending Csound a command through the virtual MIDI port. It then tells Csound how to configure Csound's patch bay to incorporate the new function. (*Micky Mic's* audio signal is sent to the computer's sound card audio input wirelessly.)

    c. Mid-level Python code continues to poll the network for changes every ¼ of a second. Since there is now a block in the network, it also initiates another loop which updates the LED values of all blocks connected to the network every .1 seconds. In this example, it will update *Micky's* LED value each cycle. The procedure is as follows:

        i. Mid-level Python code queries the high-level Python code for the LED value.

        ii. High-level Python code sends a command to Csound via the virtual MIDI port requesting the LED value.

iii. Csound returns the value to the high-level Python code through a separate virtual MIDI port

iv. High-level Python code translates the value if necessary, and sends this information back to the mid-level code. It also may signal the mid-level Python code if it determines that the block is undergoing rapid state change. In this situation, the mid-level Python code will change its behavior as explained in section 5.6.

v. Mid-level Python code sends the LED value to *Micky* via the host circuit, and tells *Micky* to light its LED

vi. Under normal circumstances, the mid-level Python code will shut off *Micky*'s LED after .1 second, then repeat this loop.

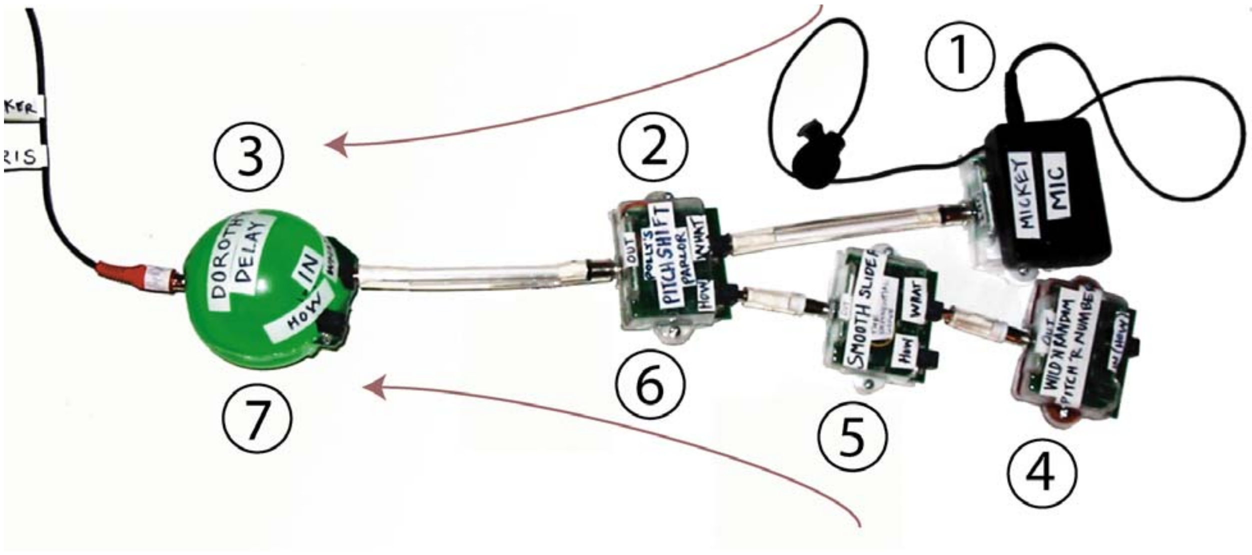


Figure 30: network demonstrating LED lighting sequence

3. The user disconnects *Micky Mic* and attaches the network shown in Figure 30.

   a. The procedure for network discovery, getting LED values and updating LEDs will be exactly the same as described with just *Micky Mic.* In the new configuration, when no blocks are undergoing rapid state change, the LED sequence shows the data flow within the network. The numbers in Figure 30 correspond to the order in which the LEDs will light. The arrows show the data flow within the network.

5.3 Block Details

SoundBlocks output their signals through a female RCA jack and input their signals through DC male jacks. Stiff connectors with an RCA plug on one side and a DC plug on the other allows users to connect blocks to each other.

A block may optionally have sensors. These sensors can be anything which connects the block to the external world, including a set of buttons, a potentiometer, a digital encoder, or any type of standard sensor such as gyroscope, accelerometer, light sensor, or ultrasonic proximity detector. It can transmit up to four dimensions of sensor data. The RGB in each block is capable of showing any combination of shades between two LED colors at a particular time.

Internally, the Atmel TINY2313 microcontroller of each block stores a type number describing the block's intended class of functions. It also contains a four-byte serial number, the number of inputs the block has, and the number of dimensions of data provided by the sensors. Assembler code in the microcontroller of each block is mostly generic: the same code is largely used for every block, independent of a block's type, serial number, number of inputs, and number of sensors. The code supports methods to read the block's various attributes, as well as commands to both control and read in the block's internal state. A few bytes in the microcontroller's EEPROM store the unique information for the block, and the generic code references this. Most of the assembler code, however, is in support of the 1-wire network protocol which fully describes all network communication and is described in section 5.4.

5.4 Network Description and Protocol

The network description in *SoundBlocks* was realized specifically for this project. It developed in discussions between MEng student Andew McPherson and I. Andrew realized the protocol, developed all of the code and circuitry to support it, designed the blocks for the developed protocol, and wrote his MEng thesis on this work (McPherson 2005). Only a cursory description of the network is provided here, as a full description of it with complete documentation of all commands and the supporting assembler code for them is available in his thesis.

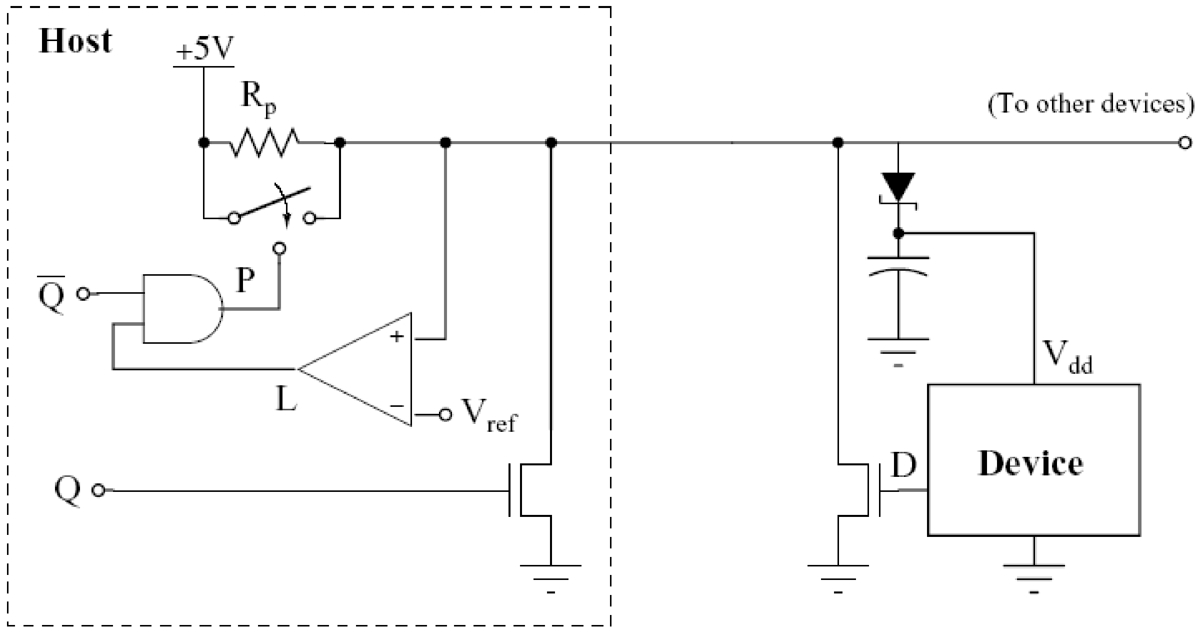


Figure 31: Network protocol (courtesy Andrew McPherson)

The network protocol provides clock, signal, and power all on wire, and supports a 57,600 baud communication speed. Although the network is implemented as host-slave, it appears to the user as peer-to-peer and is easiest to describe using parent-child terminology. The host connects directly to a computer through the serial port. All blocks are slaves. Each block contains a switch to control each of its inputs. The switch allows the block to control whether the child connected to its input has a direct connection to the host (open) or a filter signal of power only (closed).

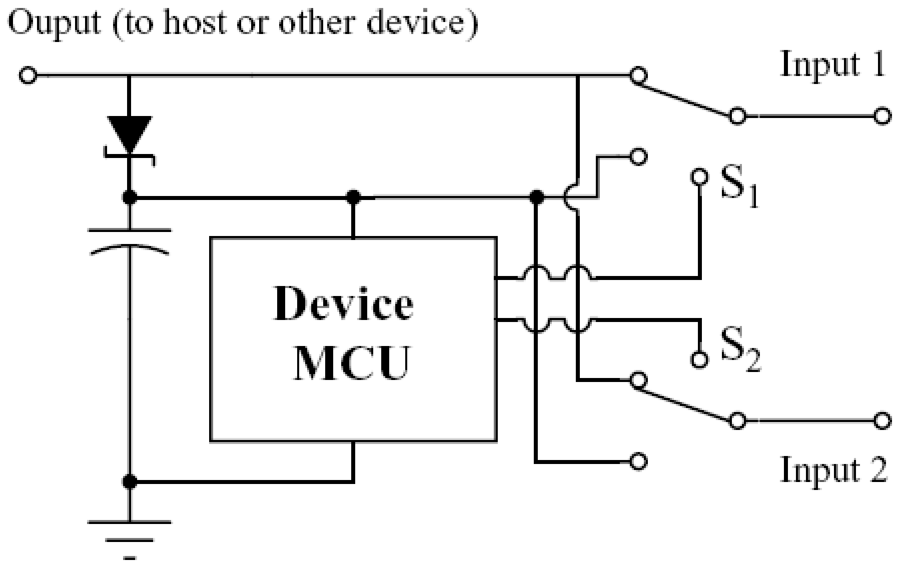


Figure 32: Slave device switches (courtesy Andrew McPherson)

When a block is initially plugged into the network, its switches default to the closed position. The host polls the network with a polling ID. Since all switches are closed in the root block, only this root block will respond to the poll. The host assigns the root block an ID number and requests that it open its first switch. The host polls the network again. The root block knows to ignore this polling request, as it has already been assigned a block ID for the currently polling ID. However, if there is a child attached to this first switch of the root block, it will see the poll for the first time and respond. The child is then assigned ID and instructed to turn it's first switch on. This depth-first-search process continues until the entire network is discovered.

At the end of network discovery, each block will have a temporary ID assigned which the computer uses to monitor and control a block's various states. For example, the host uses these IDs to control individual blocks' switches, to monitor data from sensors integrated into the blocks, and to set LED hue. There are also commands that quickly detect network changes or discover if any blocks contain unread data. The protocol allows these commands to send and receive data from all blocks at once, so it is not necessary to poll each block individually for this information.

5.5 Block Housing

Each block was housed in one of two types of casings, to see what might appeal most to children. The first casing was a pre-manufactured, light-weight, malleable playground ball of about 3.25" in diameter. We sliced the balls open, placed the circuits inside, cut holes for the connectors, then hot-glued them back shut. Because the balls were not intended for electronics, the process was a bit laborious. Moreover, the balls did not have a sturdy exterior. However, they did have a playful, simple element to them that we thought children might find appealing.

I designed the second casing using the open-source Blender 3D CAD/CAM software and the Media Lab's shop in the basement. The case was two pieces: a bottom and a covering. The bottom was a simple laser-cut base. The covering was transparent, made of PetG from a plaster mold. Because the casing was designed with the specific circuit boards in mind, assembly was quick and easy. After the circuit board and connectors were put in place, the top and bottom screwed together with two screws.

Because the second casing was transparent, users could see all of the electronics inside the blocks. Also, this second case had a more complicated shape to it, was slightly smaller, and did not roll. These differences were intentional, as I was curious how different types of learners would take to these various aspects in the user studies.

5.6 LEDs and Rapid State Change

Mid-level Python code monitors and controls when and what hue each of the LEDs in the network lights. The code supports multiple modes of operation. The high-level Python code signals to the mid-level code if there should be a mode change and, if so, what blocks are causing the change. The modes are hierarchical, arranged by priority.

Currently two modes of operation have been implemented:

1. The code defaults to mode 3, which is the lowest priority mode. It remains in this mode unless signaled by the high-level Python code to override this mode by switching to mode 2.

2. Mode 2 indicates that high-priority LED updates are needed for individual blocks in rapid state change.

To control operation in each of the modes, the mid-level code maintains two lists. The first is a list of blocks arranged to represent the signal flow between the blocks. The network in Figure 8, for example, would generate the following list: *Micky Mic, Polly's PitchShift Parlor, Dorothy Delay's Den, Wild 'N Random Pitch 'R Number, Smooth Slider, Polly's PitchShift Parlor, Dorothy Delay's Den.* This is also discussed in section 5.2. The second is a list of blocks in mode 2 operation.

In mode 3, the code cycles through the list describing signal flow at a rate of 5 blocks each second. It lights each block's LED one at a time, showing each block's current state by the LED hue. In this way, the default mode of operation shows signal flow and the relative internal state of each block to the user.

As explained in section 5.2, the high-level Python code is called every time the mid-level Python code needs to update a LED value for a block. The high-level Python code receives the value by polling Csound, then remaps it and evaluates whether or not the block should be switched to mode 2 operation. If so, it signals the block number to the mid-level Python code as mode 2. To keep power consumption within the network minimal, a maximum of two blocks can be mode 2 at a given time. If a 3rd block claims mode 2 operation, it is ignored.

If one or more blocks is marked for mode 2 operation, mode 3 operation is suspended. Instead, the LEDs for the mode 2 blocks are lit continuously, and their hues are updated every 3/100s of a second. With each update, the high-level Python code determines if the block's mode should be switched back to mode 3.

Mode 2 offers a way for users to get instant visual feedback when changing the internal state of a specific block. For example, if the microphone block is connected, it will shift to mode 2 if Csound detects active audio input. In this way, the LED color gives immediate feedback as to the amplitude of the signal the microphone is picking up. Similarly, a *Pitch 'R Number* block switches to mode 2 when its values are changing, i.e. when the user is turning its knob. Then the user can see the LED lights of the Pitch 'R Number block change in exact correspondence with the potentiometer or rotary encoder.

5.7 Csound, High-level Python Code, and the Virtual Patch Bay

Csound and the high-level Python code communicate to each other using a virtual MIDI port. However, the data they communicate to each other in no way resembles the MIDI specification. Commands from Python to Csound followed a unique protocol designed specifically for this project. The commands consisted of up to three bytes, and allowed for up to 5 parameters for each command. They followed the following format:

```
Byte 1: 0xc0    [command]       [parameter 1]
Byte 2: 0xb0    [parameter 2]   [parameter 3]
Byte 3: 0xa0    [parameter 4]   [parameter 5]
```

Csound is a compiled language. Csound run time is called performance time, since Csound is intended for music performance. Csound code primarily consists of an orchestra, and the orchestra is composed primarily of instruments. In some sense, orchestra is Csound terminology for code, and instruments is Csound terminology for classes of functions. Typically, the programmer specifies the classes, the instances of these classes, and how the instances communicate to each other at compile time, not performance time.

While it is straightforward to create instances of classes within the orchestra itself at performance time, the inherent Csound architecture does not support an infrastructure for how these instances might then be told how to communicate to each other. I wrote code to support this infrastructure, a virtual patch bay, within Csound.

The virtual patch bay supports dynamically creating and destroying functions and manipulating how these functions connect to each other, all at performance time. This gives Csound a capability previously considered possible only in Max/MSP and Pd. I know of no other examples where this has been done before in Csound. The implications for what this might allow for Csound in the future are significant.

Specifically within *SoundBlocks*, the virtual patch bay enables Python to instruct Csound to create instances, delete instances, connect and disconnect instances to each other, and send values to Python from instances. This is implemented using the virtual MIDI port.

The virtual patch bay relies on the notion of an instance number, which is similar in concept to a handle. When high-level Python code signals for Csound to create an instance, it also assigns Csound an instance number for that instance. The instance number is uniquely defined and gives both Csound and Python a way to identify this instance. Csound uses the instance number to maintain a set of arrays (called ftables in Csound) whose indices are uniquely defined by the instance numbers. These indices keep track of specific parameters for each of the instances. For example, an instance will read its input values from a specific and unique set of indices stored in an ftable called giInArray. It will write its output values to a specified location, and this location is determined by a unique set of indices stored in an ftable called giOutArray.

5.8 *Context-aware Behavior in the High-level Python Code*

The high-level Python code makes decisions regarding what a block's specific function is, what its state is, and what its default values are. These are determined based on the configuration of the network, so they are context-dependent. The intention is that the blocks might then be more intuitive and more powerful for users. I outline the specifics of what was done and the benefits it had for the system below:

5.8.1 There is no distinction between audio data and control data in **SoundBlocks**

When designing his *Music 11* sound synthesis language, Barry Vercoe established the distinction between audio data and control data (Vercoe 2005). This distinction, and the sampling rate and control rate terminology associated with them, continues to be the framework for all audio-processing languages. It is a powerful distinction for low-level processing. However, it remains confusing and problematic for inexperienced users and partially contributes to the steep learning curve associated with sound-processing languages. Indeed it is a major reason why most networks in sound processing languages will in general not create audible or sensible output. Even advanced users spend considerable effort converting between control parameters and audio parameters, sharing data between them, and choosing the appropriate primitive for the given type of data. In short, the framework distinguishing audio data from control data significantly contributes to the barriers that make sound design inaccessible to many people.

The high-level Python code in *SoundBlocks* eliminates the distinction between these two types of data for the user. It does this through a context-aware translation. The user is free to create high-level physical network constructions that do not specify data types. The code translates these into low-level virtual network constructions which do specify data types and which Csound can process directly. This mapping interprets what types of data to use where in the network, as a function of the configuration of the blocks. To understand what this means and how it was done, it is important to understand the distinctions between the two types of data.

5.8.1.1 Definitions: Audio Data, Control Data, Audio Sampling Rate, Control Sampling Rate

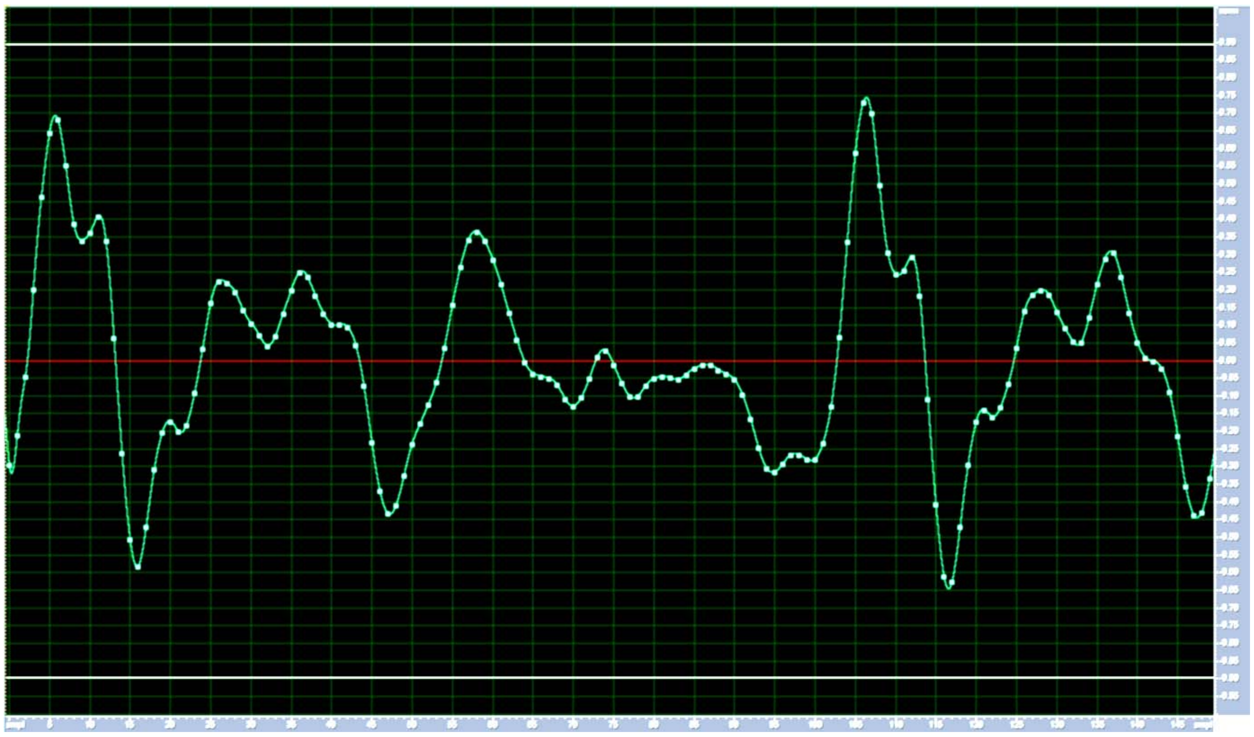


Figure 33: A 3mS of human speech as 150 audio data points

Since digital computers cannot manipulate continuous streams of data, audio must be converted to a steady stream of numbers i.e. *digital audio data*, before a computer can manipulate it. An analog-to-digital converter converts analog audio signals to digital audio data by taking snapshots of the continuous stream of audio at a fixed frequency of time --- the *audio sampling frequency*. Compact-disc-quality sound uses an audio sampling frequency of 44,100. This means it takes 44,100 snapshots for each second of audio and, thus generates 44,100 numbers to represent 1 second of CD-quality sound. Figure 33 shows 3mS of a digitized audio capturing a female voice at CD audio sampling rates.

Control data (interchangeably called parameter data) is a higher-level data type. It does not describe the raw audio. Instead it describes how the audio might be generated, altered, or manipulated by a function. For example, if we wished to reduce the pitch of the audio in Figure 33 by ½, we might send both the audio data and ½ as control data to a predefined function called "pitch." Pitch would return audio data as a result of this manipulation. In audio processing languages we can even change control data such as pitch continuously. The speed at which we send this stream of control data is typically fixed and called the *control sampling rate*.

5.8.1.2 Interchanging data in a typical sound processing environment

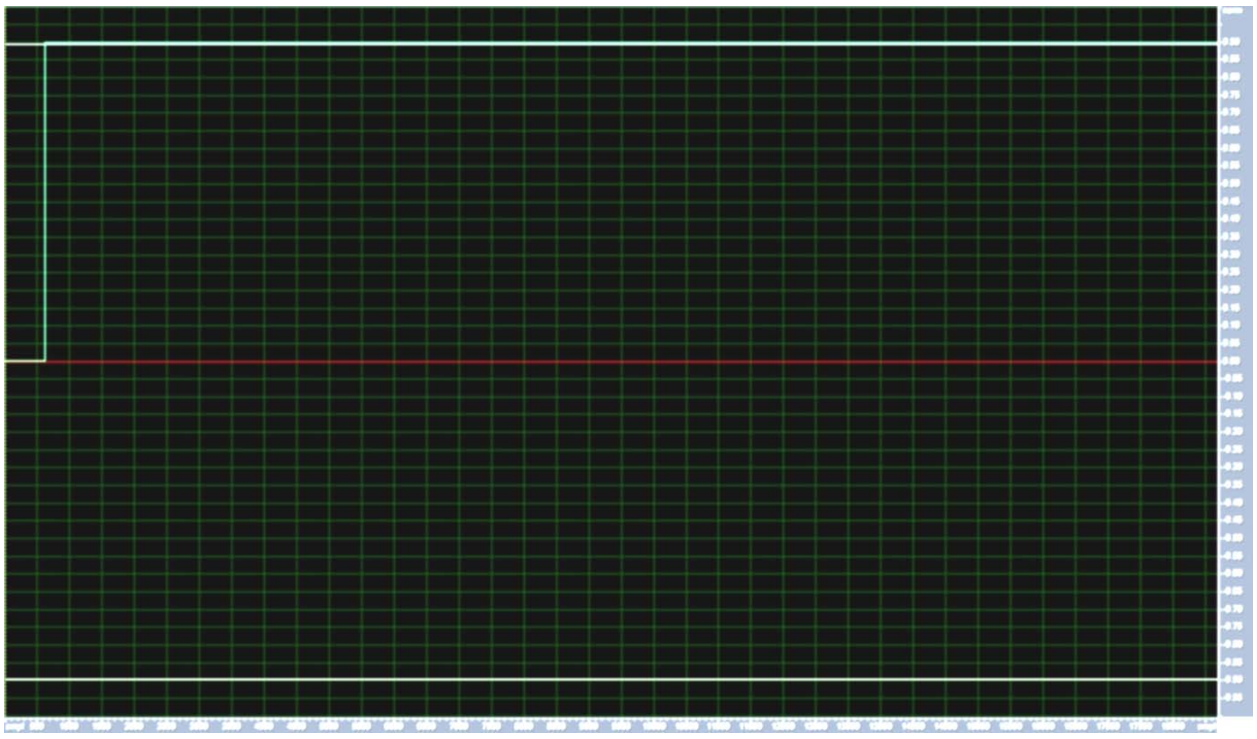


Figure 34: ½ as audio data --- DC offset

Since audio data describes what to manipulate and control data describes how to manipulate it, the two typically cannot be interchanged meaningfully. In our example, if we were to send ½ as audio data instead of control data, we would effectively have added a DC offset to the signal. At best we might he hear a click, then nothing. Figure 34 shows the corresponding signal. Similarly, sending the audio data of Figure 33 to a pitch function with some arbitrary audio signal would most likely create an unintelligible result. This is why most audio synthesis languages will not even allow such a direct conversion. The two data types have different sampling rates and different meanings. They are not to be used interchangeably.

5.8.1.3 Interchanging data in SoundBlocks

Internally within *Soundblocks*, each block can poll its output to determine whether the parent block it is connected to is expecting audio data or control data. The block may then define its behavior, send an output appropriate for the given data type, and change the expected data types for its own inputs based on this information. We can illustrate the pitch shifting scenario discussed above within *SoundBlocks* using *Pitch 'R Number, Polly's PitchShift Parlor*, and *The Sample Maker*.

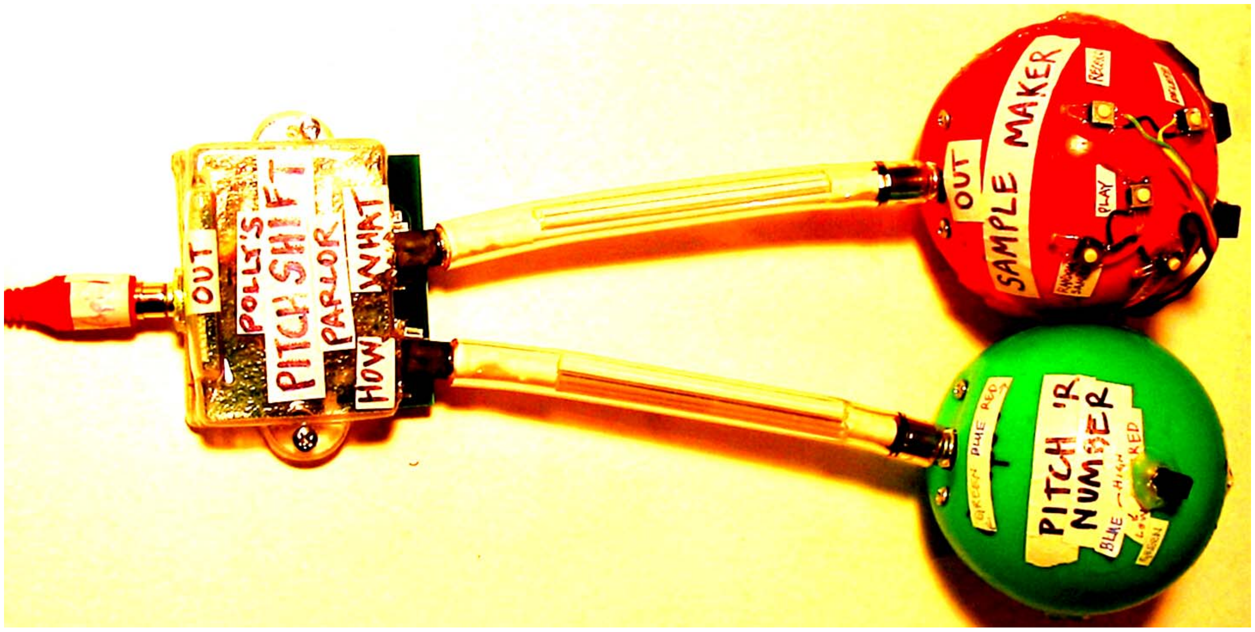


Figure 35: Pitch shifting audio from the Sample Maker

Consider a scenario where *The Sample Maker* has the stored audio and *The Pitch 'R Number* block has a value by which we wish to pitch shift this audio. We could duplicate the exact configuration described in section 5.8.1.1 by connecting *The Sample Maker* to *Polly's* "what" input and *Pitch 'R Number* to *Polly's* "how" input. This is shown in Figure 35. In this configuration, *Polly* will accept a parameter input from *Pitch 'R Number* to pitch shift the audio in *The Sample Maker*.

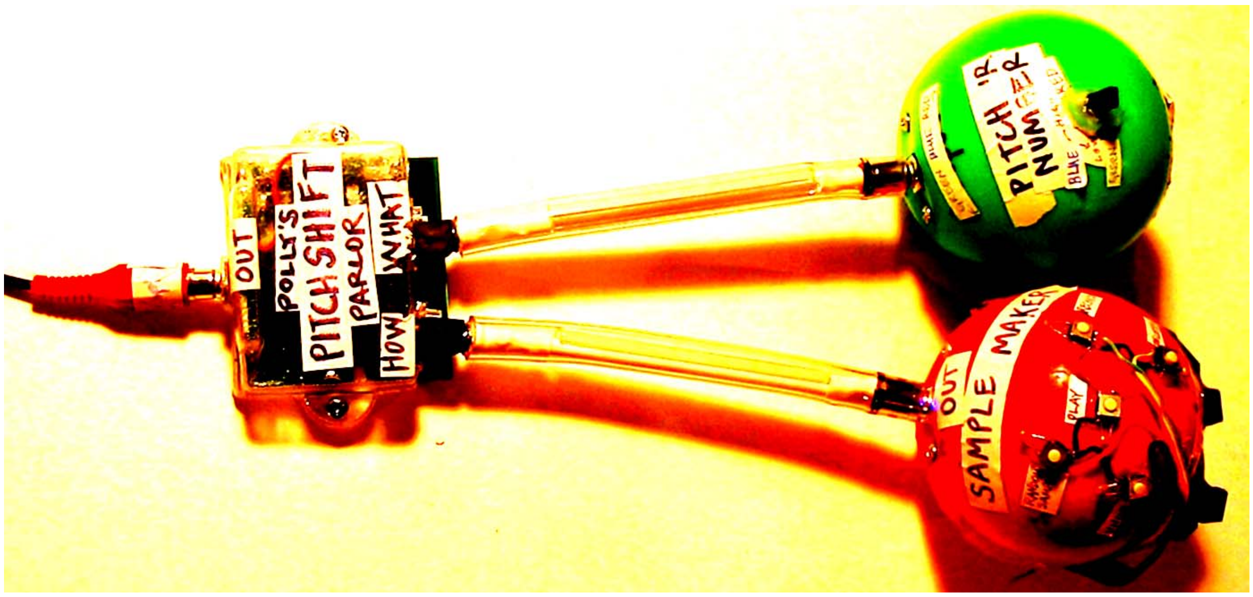


Figure 36: Pitch shifting Pitch 'R Number

What if, however, we switch *Polly's* inputs? Now *Pitch 'R Number* is connected to *Polly's* "what" input and *The Sample Maker* is connect to *Polly's* "how" input, as shown in Figure 36. If the blocks were not context aware, this would be the equivalent of trying to pitch shift an audio signal described by one number and using audio data to do it. As discussed in section 5.8.1.2, this would likely produce no audio whatsoever or worse, cause an error.

With this new configuration using the context-aware *SoundBlocks*, *Pitch 'R Number* detects that is connected to a block that is expecting audio data. Instead of outputting a number representing its parameter data like it had previously, it now switches its output to audio data of a square wave with a frequency that represents the number it was previously sending. *The Sample Maker* makes the opposite change. Detecting that its parent block *Polly* is now expecting it to send parameter data, it begins to output parameter values corresponding to the RMS amplitude of the audio signal it was previously sending. Therefore, the user will hear a square wave whose frequency changes as a function of the volume of the audio signal. The context-dependent behavior is completely transparent to the user. If the context-dependent mappings specific to the blocks are well designed, the user will perhaps find the blocks both intuitive and consistent in their behavior. Moreover, she will find them more powerful, flexible, easier to understand, and easier to manipulate than in a standard audio processing language. She does not need to wrestle with incompatible data types and their corresponding sampling speeds. Also, a greatly expanded number of networks suddenly become possible within a limited set of blocks.

5.8.2 Support default values, default states, and default behavior based on neighbor blocks

Besides being able to poll their outputs as described above, blocks also poll their inputs to determine whether or not a block was connected to them and, if so, what type of block it is. It uses this information to assume default values, default states, and default behavior. One example of this is the *Robot Combiner Diner* block.

The *Robot Combiner Diner* functions within the system as a vocoder with two inputs for audio data. It interprets one input as the carrier signal and the other as the formant. Its output is a cross-synthesis of the two with the RMS amplitude of the formant. This operation is commonly used to create robotic-sounding voices in popular music. Although it is not straightforward to explain how it works to a child or what a carrier and formant wave is, it is nonetheless a powerful tool which children could find potentially expressive. The challenge is to give the child the expressive power of this tool while at the same time making it easy for the child to both use the block effectively and understand the block's behavior.

To address this challenge, I designed the *Robot Combiner Diner* with both a default formant wave (a voice explaining a bit about the block) and default carrier wave (a metallic, consistent sound.) If no blocks are connected to its inputs, the *Robot Combiner Diner* defaults to cross-synthesizing its two default waves. If one block is connected to its input, it queries for the type of block and makes its best guess as to whether the connected block is a formant or a carrier wave. If *Micky Mic* has been attached to one of its inputs, for example, the microphone's signal is probably intended to be a formant wave. Most likely the user wants to hear his voice or a sound around him cross-synthesized with the default carrier wave. If, however, a *Pitch 'R Number* block is connected instead of *Micky Mic*, the user probably wants *Pitch 'R Number*'s square wave as a carrier to the default speech formant.

When two blocks are connected to the *Robot Combiner Diner*'s inputs, the *Combiner* queries both blocks for block type and again makes its best guess as to what the user intends, assigning one to the formant wave and the other to the carrier wave. For example, if both *Pitch 'R Number* and *The*

Sample Maker are connected to *Robot Combiner*'s inputs, *Pitch 'R Number* is assumed to be sending the carrier wave and *The Sample Maker* is assumed to be sending the formant wave. If *Pitch 'R Number* is then disconnected, *Robot Combiner* will then use its default internal carrier wave while still using *The Sample Maker*'s output as the formant wave. If *Micky Mic* is now attached where *Pitch 'R Number* is, *The Robot Combiner* will assume the microphone to be the formant wave and reassign *The Sample Maker*'s output to be the carrier wave.

The adaptive behavior of *The Robot Combiner* eliminates the need for the user to understand or even know that there is a distinction between formant waves and carrier waves. The user does not even have to be concerned with which inputs of *The Robot Combiner* they plug their blocks into, as the combiner switches the inputs as it determines is best. Although I put much thought in determining which blocks would likely be used as formants and which blocks would be used as carriers, it is worth noting that it is impossible always to know what the user intends only by looking at the block configuration. There may be situations in which the system will guess incorrectly. For example, the user might be at the beach and wish to use the sound of the waves as picked up from the microphone as the carrier for a pre-recorded sample in *The Sample Maker*. I imagine these situations to be rare and believe that the added flexibility and usability of the system as provided by this adaptive behavior more than offsets this loss of user control for the intended audience.

5.9 Specific Block Functions

5.9.1 Blocks that Generate Sounds and Values

1. **Micky Microphone:**
   - General function: send microphone data
   - Sensors: 1 - audio microphone
   - Inputs: 0
   - Output:
     - if parent requests audio data: audio from microphone
     - if parent requests control data: RMS of audio
   - LED: shows RMS of audio

- Mode 2: triggered while RMS of audio exceeds threshold

2. **Pitch 'R Number**
   - General function: generate a square wave or send a number
   - Sensors: 1 – potentiometer.
     - Turning the potentiometer clockwise sends higher values
     - Turning the potentiometer counterclockwise sends lower values
   - Inputs: 0
   - Output:
     - If parent requests audio data: square wave of frequency corresponding to the value represented by the potentiometer.
     - If parent requests control data: value represented by the potentiometer.
   - LED: value represented by the potentiometer.
   - Mode 2: when value is changing (user is turning the knob).
   - Note: there were 2 of these in the set

3. **Wild 'N Random Pitch 'R Number**
   - General function: send random values at a steady speed
   - Sensors: 0
   - Inputs: 1
     - Purpose: speed at which to generate random values
     - Type: control
     - Default value: .25 seconds
   - Output:
     - If parent requests audio data: square wave of frequency corresponding to current generated random value.
     - If parent requests control data: current generated random value.
   - LED: current generated random value.
   - Mode 2: never
   - Note: there were 2 of these in the set

5.9.2 Blocks that are Mathematic Manipulations

4. **Average (But Not Boring)**
   - General function: average out its input values
   - Sensors: 0
   - Inputs: 4
     - All inputs are identical
     - Purpose: value to average with the other inputs
     - Type: control
     - Default: none. If no block is connected to an input, that input is ignored and not averaged into the final result. If no blocks are connected, the result is the exact middle of the range of values allowed (128).
   - Output:
     - If parent requests audio data: square wave of frequency corresponding to average of the inputs.
     - If parent requests control data: average of the inputs
   - LED: average of the inputs
   - Mode 2: never

5. **Smooth Slider**
   - General function: exponentially interpolate between its stored value and the input value over a specified amount of time. Stored value is continually updated by the interpolation.
   - Sensors: 0
   - Inputs: 2
     - "What" input:
       - Purpose: provide a value to exponentially interpolate toward
       - Type: control
       - Default: Middle of the range of possible values (128)
     - "How" input:

        - Purpose: specify how much time to make it ½ way to the desired value
        - Type: control
        - Default: .25 seconds
- Output:
    - If parent requests audio data: square wave with frequency corresponding to stored value.
    - If parent request control data: stored value.
- LED: stored value
- Mode 2: never

6. **Hold**

- General function: send stored value. Update stored value with input only when button is pressed.
- Sensors: 1 – button
- Inputs: 1
    - Purpose: provide value to update stored value.
    - Type: control
    - Default: Middle of the range of possible values (128)
- Output:
    - If parent requests audio data: square wave with frequency corresponding to stored value.
    - If parent request control data: stored value.
- LED: stored value
- Mode 2: never

5.9.3 Blocks that Manipulate pre-existing Audio (named after places and rooms)

7. **Dorothy Delay's Den**

- General function: delay a signal
- Sensors: 0

- Inputs: 2
    - “Signal” input:
        - Purpose: provides signal which is to be delayed
        - Type:
            - Audio data if parent block expects audio data
            - Control data if parent block expects control data
    - “How” input:
        - Purpose: accepts a control parameter specifying delay time from 0 to 2.5 seconds.
        - Type: Control data
        - Default value: 1.2 seconds
- Output: Delayed signal
- LED:
    - If input is control: value of input
    - If input is audio: RMS of input
- Mode 2: never

8. Robotic Combiner Diner

- General function: phase vocoder. Cross synthesizes the inputs, and returns the resulting signal with amplitude of the formant signal.
- Sensors: 0
- Inputs 2:
    - Both inputs appear identical to the user
    - Purpose: carrier and formant waves
    - Type: audio data
    - Internally, chooses formant and carrier based on input blocks connected to it. (See section 5.8.2)
    - Has default formant and carrier waves
- Output:
    - If parent requests audio data: audio from cross synthesis
    - If parent requests control data: RMS of audio from cross synthesis

- LED: RMS of cross synthesis
- Mode 2: never

9. **Polly's PitchShift Parlor**

- General function: pitch shift to a maximum of 1 octave up or down, using phase vocoder
- Sensors: 0
- Inputs: 2
    - "What" input:
        - Purpose: accepts signal to pitch shift
        - Type: Audio data
    - "How" input:
        - Purpose: accepts parameter specifying amount of pitch shift
        - Type: Control data
        - Default: No pitch shift
- Output:
    - If parent requests audio data: audio of resulting pitch shift
    - If parent requests control data: RMS of audio of resulting pitch shift
- LED: RMS of audio of resulting pitch shift
- Mode 2: never

10. Volume

- General function: set volume of audio
- Sensors: 0
- Inputs: 2
    - "What" input:
        - Purpose: provide signal for volume adjust
        - Type: audio
    - "How" input:
        - Purpose: provide parameter for volume adjust
        - Type: control

        - Default: 1

- Output:
    - If parent requests audio data: audio
    - If parent requests control data: RMS of audio
- LED: RMS of audio
- Mode 2: never

Special Function Blocks

11. Chris the Speaker

Chris the Speaker is not actually a block but the name of the RCA cable that comes out of the host circuit. It functions as the root of the network. All network flow must eventually terminate in *Chris the Speaker*. This cable expects audio-parameter data in its "what" input. (The name "Chris" does not have a special significance, except that it is politically correct since it is gender-neutral. Recall from section 4.2.1 that audio-parameter data is data which appears to be an audio signal to the user but internally to the system is parameter data signifying audio.)

12. Ask Me

The *Ask Me* block is a helper block that can be used as a dictionary to explain the other blocks. To avoid confusing the user, this block will function only when connected directly to *Chris the Speaker*. If plugged in elsewhere, the system says "I can help you only when connected directly to *Chris*."

When the user plugs the *Ask Me* block into *Chris*, *SoundBlocks* says, "plug a block in and I'll have it tell you how it works." The user can then plug any block in and will get a very short story/explanation about what the block does and how it might be used in the network.

13. The Sample Maker (See Figure 9).

The Sample Maker functions as a digital recorder and sampler. It can record up to 16 samples of a total of 6 minutes in length. In normal operation, the sampler has two inputs: “speed” is on the left and “trigger” in on the right. “Speed” sets the sampling rate of playback. “Trigger” triggers a sample based on changes of value from the input block.

Internally, the sampler assigns each sample a range of values. When it receives a new value from the trigger input, it plays the sample assigned to the range corresponding to the new value. For example, if two samples have been recorded, the sampler assigns values 0-127 to the first sample and 128-255 to the second sample. At some point, a block connected to the trigger input might send a new value. In this example, if the new value is below 128, the first sample will be played. Otherwise the second sample will be played. The sample will be played to completion unless a new sample is triggered, in which case the original sample is stopped and then the new sample starts.

The Sample Maker has 5 buttons to support recording and reviewing samples: “record,” “play,” “delete,” “undo,” and “random samples.” To record samples, the Sample Maker must be the first block connected to the network and no blocks can be connected to the “trigger” input. This is an intentional constraint to make operation more understandable to the user. If it weren’t the case, it could be confusing for users because samples could be triggered while the user is trying to record samples.

Before recording, the user connects *The Sample Maker* to a block whose output she wishes to record. (See Figure 9). Typically, this might be *Micky Microphone*. She then presses “record” and records the sample. She hits “record” again to stop recording. The recording is automatically normalized and beginning silences are chopped off.

The Sample Maker maintains internally a list of all of the samples which have been recorded, and a pointer to a current sample within this list. The user can cycle through the samples one at a time by pressing “play.” She can delete and add samples anywhere in the list with the “delete” and “record” buttons. There is also an “undo” button which can undo the last operation. If a user presses “undo” after hitting the “delete” button, the sample will be restored. If a user pressed "undo" after hitting the "play" button, the sampler will step backwards through the sample list.

Last, if the user is not feeling especially creative, she can hit the "random samples" button. This will record a random sample into the Sample Maker from a list of interesting and varied sounds internal to the system.

If *The Sample Maker* is not the first block in the network and no blocks are connected to its trigger input, it will cycle through all of its recorded samples, one at a time.

Since *The Sample Maker* is a more complex block for the user, recorded voice files explain mistakes to the user. For example, if a user presses a button when the Sample Maker is not the first block in the network, a voice will respond "my buttons work only when I am directly connected to Chris."

5.10 Rationale for Implementation

5.10.1 The blocks are simple. All processing happens in a Central Processing Engine

Internally, *SoundBlocks* act as a tangible interface to a central processing engine. To the user, however, the blocks appear to perform the audio functions themselves. This is the same paradigm used in the first generation of *SoundBlocks*. It offers several advantages when compared with the more obvious choice to have the audio processing calculated within each block itself. These advantages include expense, flexibility, and adaptive behavior, and they are outlined below.

5.10.1.1 Avoid an expensive DSP in each block

Blocks will each need a microcontroller to support the basic functions needed within the blocks. However, we wish to keep the microcontroller as simple and inexpensive as possible. Unfortunately, digital audio processing is computationally demanding and is not possible in real-time using a simple microcontroller, even for mediocre audio quality. Real time manipulation requires a specialized Digital Signal Processing (DSP) chip. And, because of its specialized nature and powerful number crunching capabilities, a DSP chip is significantly more expensive than a simple microcontroller. If audio digital processing were to happen in each block, the cost of using a DSP over using a simple microcontroller scales the price of parts for each block by a factor of 10. With multiple blocks involved, it would not be possible to keep the product affordable.

5.10.1.2 Avoid an unnecessarily complex and expensive high-speed network infrastructure

Necessitating digital audio processing within the blocks also adds expense to the network infrastructure. If the blocks manipulate the audio, the network must support high-speed audio data transmission rates. High-speed transmission adds complexity and therefore expense.

5.10.1.3 Make the system design as flexible as possible

If each block were hard coded to perform a specific audio function, flexibility iterating the design of the system becomes more difficult. Every time a block function needs to be tweaked or a specific function needs to be rewritten or replaced, the block has to be opened and its microcontroller reprogrammed. Additionally, code on specific blocks could be challenging to debug, as each block would not have a supporting debugging environment complete with screen, keyboard, mouse and a windowing environment in which to explore the code and its behavior.

By keeping the processing outside of the blocks, each block need only store a device type, serial number, and generic code. The device type and serial number uniquely describe the block and its function. The generic code supports the network infrastructure and various sensors that might be integrated to it.

In this scenario, all of the blocks can be programmed essentially to be exactly the same. Also, since no processing happens within the block itself, it is easy to change the entire behavior of the system just by changing the centralized engine. In fact, by only changing the engine it is completely possible that SoundBlocks could become VideoBlocks or TangibleBlocks, used to manipulate video or any other type of tangible network a designer would wish to construct.

5.10.1.4 Make the environment as adaptive to the user as possible

For a system to make the best choices of adaptive behavior within a network, the system will need both representation and control of the entire network itself. This is most practically done with a centralized network. If each block attempts to control its own adaptive behavior but no one system is controlling the overall adaptive response of the network, the possibilities for adaptive behavior are limited or become unnecessarily complex.

5.10.2 Csound

Barry Vercoe has recently ported a version of Csound called Extended Csound (XTCsound) to the ADI Blackfin on a UCLinux host. This port shows great potential for future development with *SoundBlocks*. The Blackfin is a relatively inexpensive DSP and can be operated on batteries. It is easy to imagine a system where all central processing is supported on a Blackfin running XTCsound and supporting multiple channels of audio, each being digitally manipulated independently.

While Pd is also an excellent choice and has also been ported to portable platforms, the final iteration of *SoundBlocks* does not use Pd. As pointed out earlier, the visual elements of Pd are not necessary and can be, in fact, distracting to users. Pd can be operated without its GUI, but much of the advantages it previously offered are then lost.

Because Pd is interpreted instead of compiled, patches can be created, destroyed, and routed on the fly in performance time. At first it appeared that Csound, a compiled language, would not be able to also support this important functionality. However, it is possible to maintain a complete virtual instrument and patchbay system within Csound without any recompiling. *SoundBlocks* demonstrates the first such virtual patchbay implemented in Csound of which I am aware. This seems like a unique use of Csound and also suggests great possibilities for how Csound might be used in the future.

Csound, in active development since 1985, supports a rich set of opcodes, which makes prototyping rather complex audio manipulation algorithms relatively easy. Furthermore, some have suggested that Csound draws less CPU than Pd given the same set of manipulations, since Csound is compiled and Pd is interpreted. This has not been verified.

6. SoundScratch

6.1 Design Guidelines

We considered many possible designs when evaluating how to integrate digital sound manipulation into *Scratch*. Below is a list of the principals that guided the final design

6.1.1 Focus on the procedural elements, not the data elements

Scratch, influenced by LOGO, is a procedural language. Users write scripts by interlocking blocks together. From the user's perspective, the blocks do not send messages to each other. Rather, each block is executed, one at a time. Contrasting this, low level digital sound manipulation is usually described as a network of operations which each send data to each other. Indeed advanced users find a great deal of power in manipulating the connections between these operations. But this power comes at a cost. Figure 37 shows how even simple sound manipulations can create network relationships that seem complex and difficult to understand. This patch is an algorithmic note generator that I wrote.

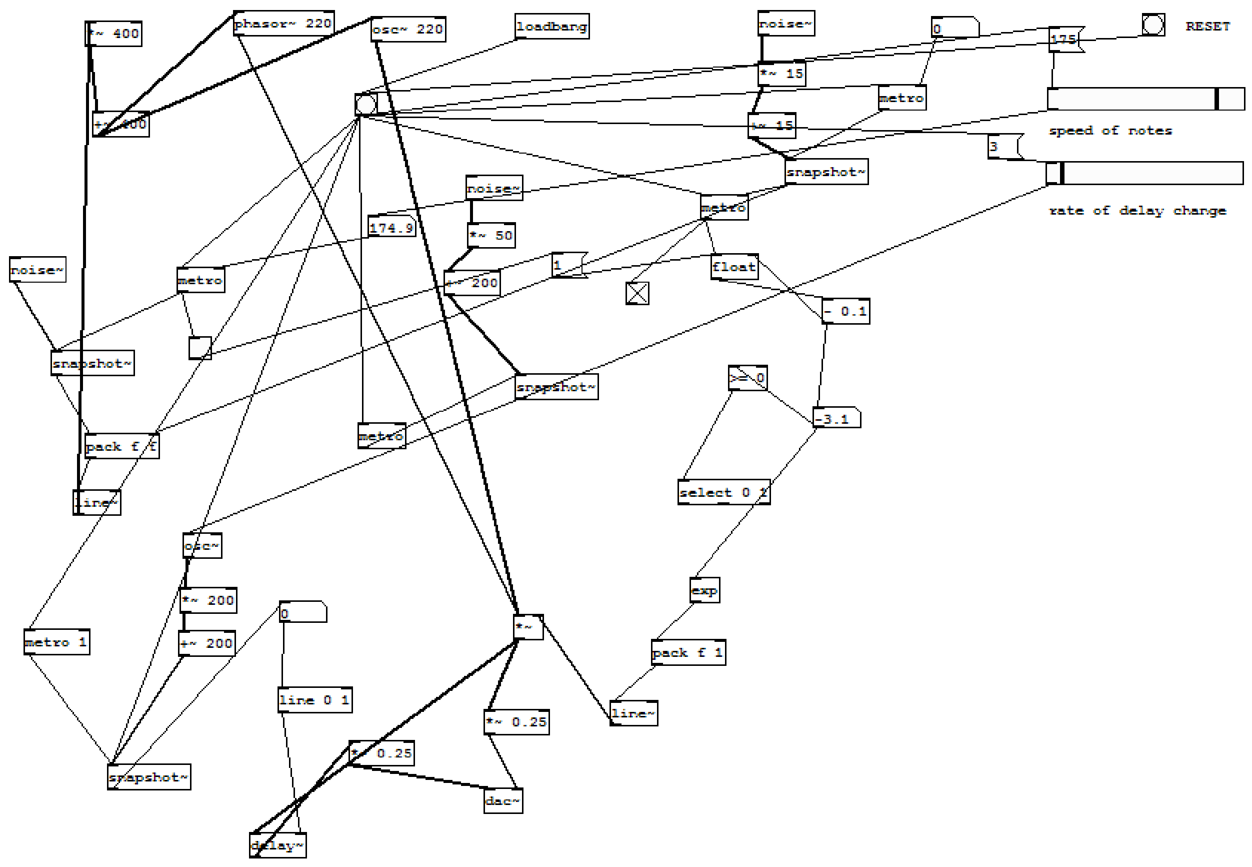
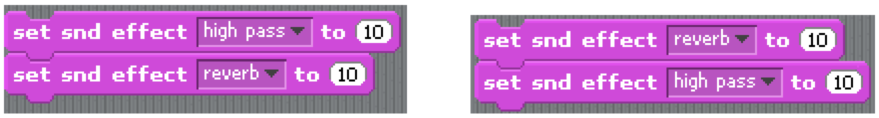


Figure 37: An Example Patch in Pd

While there is power and flexibility in allowing any possible data flow to be expressed, most data connections do not make sense. The good news is that, for a typical set of operations, a standard data flow can be predicted. Therefore, it is not necessary, and may perhaps be a hindrance, for beginning users who wish to digitally manipulate sound to have to also describe the relationships between the various sound operations. Dataflow descriptive networks are also not a currently supported framework in *Scratch*, and adding this framework clouds the streamlined focus *Scratch* currently provides to help users get started using *Scratch*.

Therefore, in *SoundScratch* the data flow has been hard-coded, hidden, and cannot be changed by the user. The order of blocks only changes the order of when values are set, not the order of when the manipulations are processed. As an example, reverb is always processed within the system after the high pass filter. This is the case independent of the order in which the blocks are executed.



```
set snd effect high pass to 10
set snd effect reverb to 10
```

```
set snd effect reverb to 10
set snd effect high pass to 10
```

Figure 38: High-pass will be processed first in both of these configurations

In Figure 38 execution of the first and second block configuration will be almost identical. In the first case, the high pass filter will receive the value 10, then reverb will receive the value 10. In the second case, first the reverb will receive the value 10, then the high pass filter will receive the value 10. In both cases, however, the high pass filter will process the audio data before reverb, because this is hard-coded within the system and cannot be changed by the user. The choices for the order in which audio data is processed were based on common practice. For example, most audio networks, like *SoundScratch*, process reverb as their last operation before the resulting output.

6.1.2 Make it easy to create data manipulations that are familiar to youth.

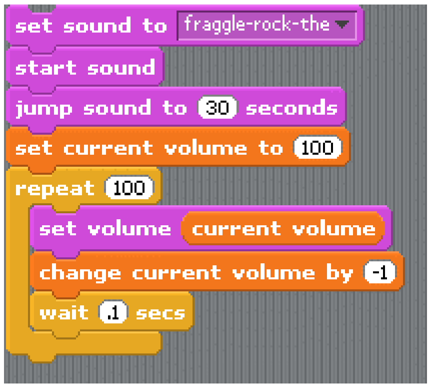
Certain digital audio effects are very common in popular music. These include scratching, stuttering, echo and reverb effects. Blocks need to support functions that lend themselves to writing scripts to create these effects. These effects may be the first manipulations that youth will try. If the system can support these effects, the youth may then explore the system further to discover new and unique manipulations that are personal and meaningful to them.

6.1.3 Support all effects in real-time

So youth can interact with the effects as both performer and designer, the system must support real-time manipulation of all audio. This means that the user should be able to apply any combination of effects to audio that is streaming in real-time and then hear the result with minimal latency. This use is demonstrated in Figure 19 and Figure 20.

6.1.4 Do what the user probably intends, not literally what the primitive means

Audio manipulations, such as pitch shifting or band-pass filtering are rarely applied directly without various smoothing and compensation operations. Without these operations, results may not be what the user intended. For example, consider the following script:

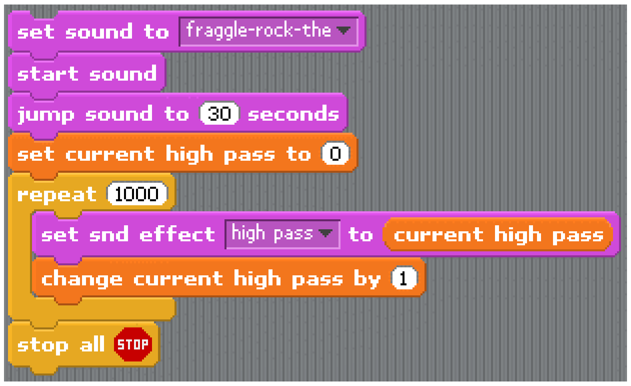


```
set sound to [fraggle-rock-the ▾]
start sound
jump sound to (30) seconds
set current volume to (100)
repeat (100)
    set volume (current volume)
    change current volume by (-1)
    wait (.1) secs
```

Figure 39: A Fade Out?

In most cases, the user actually intends a 10 second fadeout of the selected sound, starting 30 seconds into it. If volume were to be interpreted literally, however, the user would hear a click every .1 seconds as the volume were suddenly changed. This clicking, often called "zipper noise," is a common unintended artifact in digital sound manipulation. However, *SoundScratch* will not process the volume changes suddenly. Instead, it will smooth out these changes. The result will be what we guess the user intended.

As another example, consider the script for high-pass filtering in Figure 40:



```
set sound to [fraggle-rock-the ▾]
start sound
jump sound to (30) seconds
set current high pass to (0)
repeat (1000)
    set snd effect [high pass ▾] to (current high pass)
    change current high pass by (1)
stop all
```

Figure 40: High pass without compensation?

If a high-pass filter were applied without readjusting the RMS of the audio after the filter, Figure 40 would result in a gradual fade of the audio itself, and would also suffer from zipper noise. We eliminate the zipper noise by smoothing the changes applied to the filter as we also did in Figure 39. Still, the lows will be gradually removed, but the highs will not be accentuated to compensate. This is unlikely to be what the user wants. Therefore, we readjust the RMS of the resulting signal to be equivalent to the RMS of the original signal. Most likely, this is what the user intended.

6.1.5 Choose measurements and scales that minimize conversion

Any user of Csound, Pd, or Max/MSP will tell you that much of their code is merely converting from the units of one function to the units of another. Furthermore, some functions are best understood with logarithmic or exponential scales, such as frequency and amplitude. Other functions are best understood with linear scales, such as time. The *Scratch* user should not be burdened in any way with these details. All conversions should be assumed internally and hidden from the user.

6.1.6 Support multiple concurrent manipulations of sound

Scratch scripts do not support branching code into multiple concurrent operations. At the same time, letting users manipulate only one sound in one way at a time may severely limit personal expression. However, *Scratch* supports multiple sprites and each sprite can support its own independent script. By extending the sprite analogy to include sounds, multiple sounds manipulated independently can be implemented in a manner consistent within the *Scratch* framework. This was demonstrated in Figure 20.

6.1.7 *Build upon the already existing visual framework of* Scratch

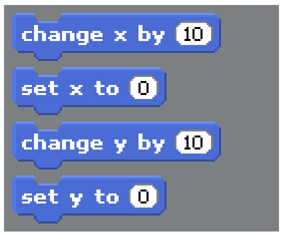

```
change x by 10
set x to 0
change y by 10
set y to 0
```

Figure 41: Blocks that manipulate visual Sprites in standard Scratch

Figure 41 shows a set of blocks used to move visual Sprites in the standard *Scratch* environment. Blocks that involve audio manipulation should build upon the analogies created in the visual primitives. For example, as x and y coordinates manipulate visual elements, perhaps pitch and tempo manipulate audio elements. Similarly, as size is an attribute to visual manipulation, volume is an analogous attribute for audio manipulation. In this way, the *Scratch* extensions for audio manipulation will feel natural and intuitive to the user, as they have analogies from the *Scratch* environment the user is used to. This provides a framework in which to expand a user's understanding of sound.

6.1.8 Require effortless installation

If we hope for individuals and perhaps even communities to try *SoundScratch*, then we should not burden them with difficult installations that require computer knowledge. Therefore, we cannot depend on specialized audio drivers, such as ASIO, and we cannot use virtual MIDI ports. Both ASIO drivers and virtual MIDI ports require installation and configuration. Then Csound has to be configured to recognize them.

Instead, we opted to use the standard Microsoft Windows drivers. The choice cost us an inherent drop in performance. Standard Microsoft Windows drivers add latency to the real-time effects and buffering is not as robust when compared to ASIO, so some clicks can occur as the computer multitasks.

Unlike *SoundBlocks*, which uses virtual MIDI ports to send and receive messages to and from Csound, we instead use pipes. This means that Squeak and Csound communicate information between each other by writing inside files. Coordinating this in design is tricky as each has to open the file it writes to before the other reads from it. Also, it required that Csound source code be recompiled to include flushing commands so Squeak could see the information Csound was writing. A Windows script handles all of the details, including killing the Csound task when Scratch quits. The end result is a robust system that requires no configuration and is easy to execute.

6.2 Implementation

The underlying code for *Scratch* has been written in Squeak, an open source version of Smalltalk. This Squeak code was extended to make calls to Csound, the sound-processing engine. Like *SoundBlocks, SoundScratch* uses instance numbers as a way for both Csound and the calling code to keep track of the independent sound manipulations.

When a new sound is created or loaded, *SoundScratch* writes the sound to a temporary dictionary, and then loads it into a Csound ftable. It then updates a dictionary that maps this sound to the corresponding Csound ftable number. As sounds are started, stopped, and modified, *SoundScratch* sends the command with the instance number and ftable information if necessary to Csound.

Within Csound, one rather large instrument, the main instrument, has all possible effects described. *Scratch* sends Csound commands to initialize instances of this instrument. Within this instrument, Csound code skips the effects that are not being used and processes the effects that are being used. Information is passed to the specific instrument that has been assigned to each command. Command instruments are used to send commands or change parameters in already existing main instruments. Like with *SoundBlocks,* this information is sent to the main instrument by specifying indexes in arrays, defined uniquely by the instance number of the main instrument.

7. Evaluation

7.1 SoundBlocks

7.1.1 Overall Evaluation Strategy

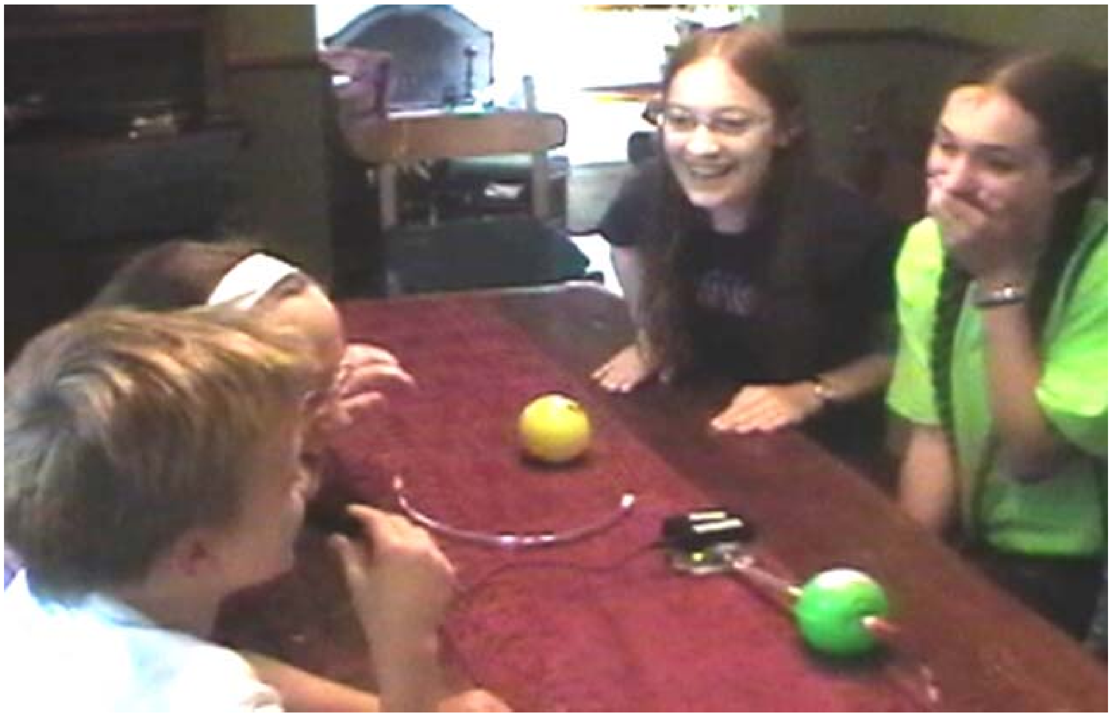


Figure 42: Singing with SoundBlocks

We evaluated *SoundBlocks* for usability, understandability, and expressiveness for youth ages 10-15. In evaluating understandability, we tried to determine how well the youth understood the functions of the blocks, the behavior of the network, and the sounds they were creating.

We arranged for youth to play with *SoundBlocks* in groups of 2-5 on 8 separate occasions. Sessions were between ½ hour and 2 ½ hours. All evaluations took place with at least one music educator present. Evaluation session locations were widely varied. Some took place in the kids' homes, some at a day camp, and some in my own home. The sessions took place at different times of day and between a variety of scheduled and unscheduled activities. The kids, their backgrounds, and my relationship to them were also widely varied. Some kids were children of friends of mine, and some I didn't know and didn't know me. Some had strong musical backgrounds and some professed to hate musical instruments. Sometimes sessions included youth who had seen *SoundBlocks* previously.

For the evaluation session, we organized a 4-part tutorial, designed to be 20 minutes in length. In the tutorial, the children took turns playing with the network as I guided them. As the youth continued to explore the blocks, music educators and I observed. At times we posed challenges to the children related to their interaction with the blocks. Typically, the challenges were to create this or that network or this or that type of manipulation or sound. The children's responses to these challenges helped us comprehend their understanding of the blocks and of the sounds they were creating.

During the sessions, we also asked youth questions about themselves. The questions included what their own interests were, what their favorite subjects were in school, what they did with their friends, what they liked and disliked about the blocks, and what their favorite block was and why. Depending on their answers, we sometimes asked follow-up questions. The purpose of these questions was to understand more about the youth, have them feel comfortable with both the blocks and with us, and gain a better understanding of the youth's perspective of the blocks.

A few of the sessions were videotaped. Some of the resulting sounds from the sessions have been preserved as audio. Both this video and audio have been reviewed to determine the expressiveness of the system as well as to better define what is possible for children to create within the system in a short time frame. In total, we have approximately 2 ½ hours of raw video footage and 2 ½ hours of audio.

7.1.2 Tutorial introducing the blocks to Youth

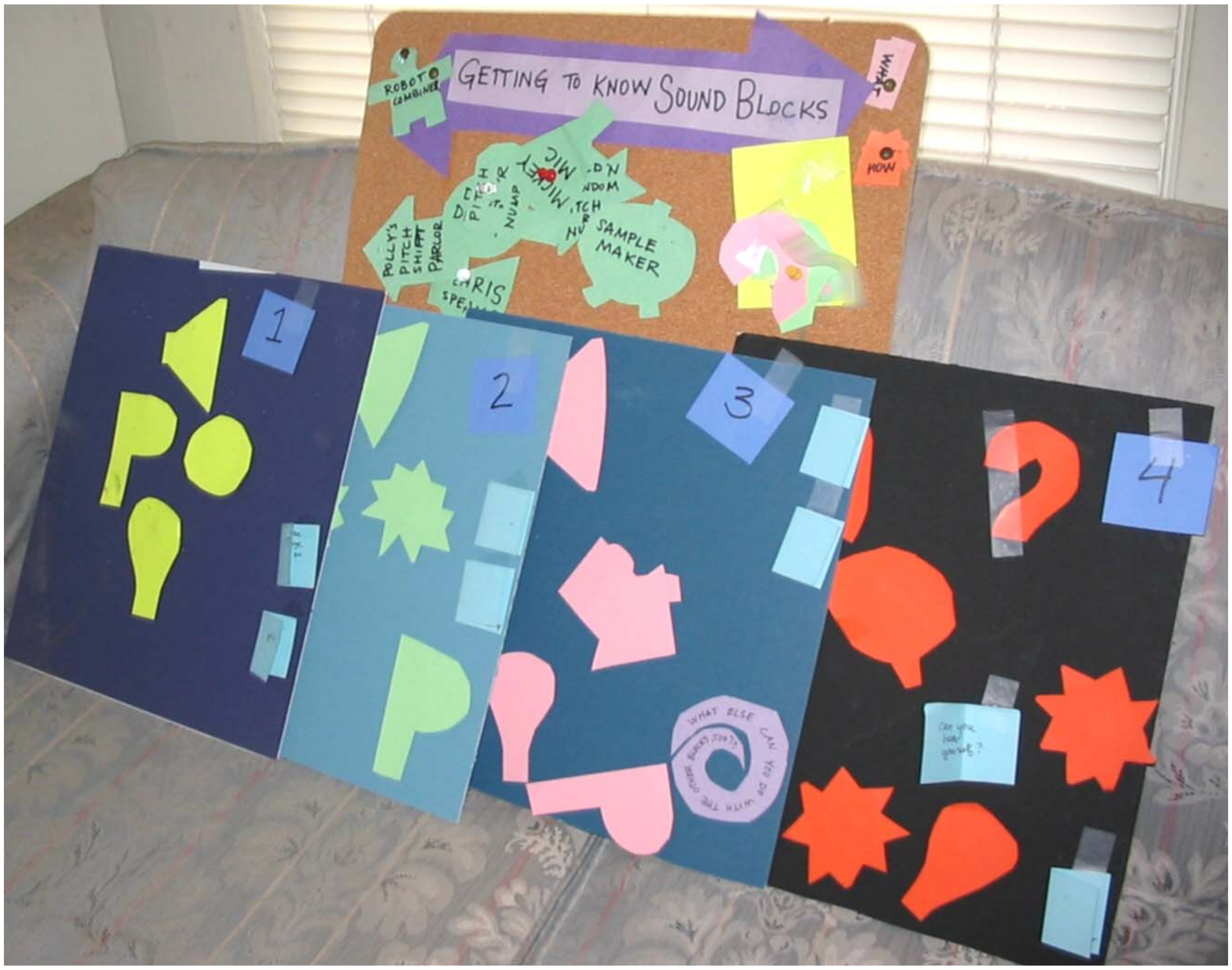

GETTING TO KNOW SOUND BLOCKS
ROBOT COMBINER
WHAT
HOW
MICKEY MIC
POLLY'S PITCH SHIFT PARLOR
SAMPLE MAKER
CHRIS SPE
1
2
3
4
WHAT ELSE CAN YOU DO WITH THE OTHER BLOCKS TODAY?

Figure 43: Visual Aids for *SoundBlocks* tutorial

I worked with a music educator to develop a short tutorial of four lessons that together could introduce the blocks to small groups of children. The four lessons were designed to take 20 minutes. They broke down as follows:

7.1.2.1 Micky Microphone and Dorothy Delay

- *Purpose: A gentle introduction to* SoundBlocks. *Explain that the blocks are used to manipulate the sounds around us. Create a simple network to manipulate sound in real-time and have the youth interact with it.*
- Introduce *Chris the Speaker*. We hear whatever we hook up to *Chris*.

- Introduce *Micky Microphone*. Have a kid connect *Micky* to *Chris*. Explain the concept of "in" and "out" as it relates to the network.
- Have a kid connect *Dorothy Delay's Den* between *Chris the Speaker* and *Micky Microphone*. This demonstrates how to create a network. It also introduces the distinctions in input between "how" and "what."
- Introduce *Pitch 'R Number*. Show how the delay changes as a function of *Delay's* "what" input using *Pitch 'R Number*.

7.1.2.2 Pitch 'R Number and Wild 'N Random Pitch 'R Number

- *Purpose: Demonstrate that we can play with the network to discover the functions of various blocks. Show two basic primitives:* Pitch 'R Number *and* Wild 'N Random Pitch 'R Number.
- Show how *Pitch 'R Number* generates audio when connected to *Chris*.
- Make up games so kids use *Pitch 'R Number* to emulate *Wild 'N Random Pitch 'R Number*.
- Introduce *Wild 'N Random Pitch 'R Number:* have a kid connect this block directly to *Chris*.
- Have a kid plug *Pitch 'R Number* into *Wild 'N Random Pitch 'R Number*.
- Optional: have the youth plug two *Wild 'N Random Pitch 'R Numbers* together.

7.1.2.3 Polly's PitchShift Parlor and the Robotic Combiner Diner

- *Purpose: Introduce some of the blocks that manipulate sound. Have the youth feel comfortable playing with the sound manipulators named after rooms:* Dorothy Delay's Diner, Polly's PitchShift Parlor, *and* The Robotic Combiner Diner
- Have a kid connect *Polly's PitchShift Parlor* to *Chris the Speaker*. Ask the kids why they do not hear anything.
- Have a kid connect *Micky Microphone* to *Polly's* "what" input. Now we hear the output of the microphone, but the pitch is not shifted. Ask the youth why this might be and what they might try.

- Have a kid connect *Pitch 'R Number* to *Polly's PitchShift Parlor's* "how" input. Have the youth play with this configuration.
- Have a kid replace *Pitch 'R Number* with *Wild 'N Random Pitch 'R Number* and/or replace *Polly's PitchShift Parlor* with *The Robotic Combiner Diner*

7.1.2.4 The Sample Maker

- *Purpose: Introduce* The Sample Maker. *Through the tutorial, have the youth practice learning about the blocks through exploration.*
- Explain the purpose of *The Sample Maker.*
- Have a kid connect *The Sample Maker* directly to *Chris.*
- Have a kid press the *record* button of *The Sample Maker.* The block will respond by telling them they need to connect a block to its *record* input.
- We discuss what we want to record. Probably it will be *Micky Microphone.* A kid connects *Micky* to *The Sample Maker's* "record" input.
- We record some samples. We practice reviewing and deleting samples, and choosing from the random sample list.
- We hook up a simple network that triggers the recorded samples randomly. We connect a *Pitch 'R Number* block to *The Sample Maker's* "speed" input to change the speed at which these samples are played.

After completion of the tutorial, the children were encouraged to play freely with the blocks as guided by their own understanding and their own imaginations. The music educators and I were available as a resource and to observe their interaction and creations. This time of free play had no specified time window. It was as long as the youth wished to play or until another event dictated that they leave. In general, the music educators and I tried to keep our interference during free play at a minimum. There were even periods where we intentionally left the youth alone to explore the blocks without supervision.

7.1.3 Tutorial Presentation

In the evaluation sessions, I introduced the blocks using three different approaches. In the first approach, I stepped them through the short tutorial outlined above, making sure the tutorial was complete within 20 minutes. Sticking to this timeframe was not always easy. Typically, youth would become excited about the blocks right at the beginning of the tutorial and would actually want to grab the network away from me so they could start playing with it themselves. While I allowed them to experiment to a limited degree, I also did not hesitate to stop their play so I could remain on schedule. The result was an interactive tutorial that was at the same time completely in my control. During the tutorial, the youth were not allowed to freely play with the blocks.

In the second approach, I followed a much looser structure. I used the tutorial only as a way to initially introduce the blocks. As the youth became actively involved, I would allow them to completely take the blocks over before the tutorial was complete. This way their own explorations would largely guide their initial introduction to the blocks. If they got stuck or asked me questions, I would perhaps borrow from the tutorial to help them discover more about the blocks. This approach meant that we did not always complete the tutorial, and if we did, then it would not be within the 20-minute time frame. It blurred the distinction between tutorial and free play.

Sometimes an evaluation session included a child who had seen the blocks before. When this occurred, I tried a third approach. The third approach included no tutorial and as little direction as possible from me. Instead of providing continual direction, I asked the child who had seen the blocks before if he wanted to work with me to introduce the blocks. I would let this child lead an introduction in any way he wished, to the degree to which he was comfortable doing so. If I sensed that the child was struggling with the explanations or there was some confusion within the group, I would offer guidance.

7.1.4 Observations

Figure 44: Experimentation with *SoundBlocks*

This section is a compilation of the observations made by music educators and by me from the evaluation sessions.

7.1.4.1 **"Into Computers" vs. "Artistic"**

During the evaluation sessions, children tended to describe themselves as either "into computers" or "artistic," and few children described themselves as both. The youth into computers did not show a better understanding of the network behavior. Moreover, the self-described artistic children created some of the most interesting sounds and structures when using *SoundBlocks*.

The most interesting sounds came from a group of three girls ages 12-14, all friends, and all musically inclined. One of them had played with the blocks previously, and she led the initial introduction. Already, with just *Dorothy Delay* and *Micky Microphone*, the group created activities I had not thought of. They sang a call and response song where they sang the call and the blocks "sang" their response. Later, the girls sang in harmony and recorded this in the sampler maker. They then used the robot combiner to cross-synthesize this recorded harmony with the microphone input and delay. They spoke into the microphone and also tried to sing along with it. These kids appeared to me the most inspired of any that worked with the blocks. They played with it for 2 ½ hours, at which point I had to ask them to leave because I had other appointments.

However, non-musical kids also enjoyed exploring the blocks and created interesting sounds. Cody, a 12-year-old boy, stated at the start of his tutorial that he hated musical instruments. In spite of this proclamation, he became very involved in the process of creating sounds with the blocks.

7.1.4.2 Understanding

In general, children ages 14 and older could completely comprehend the signal flow within the network. Children ages 8-10 could understand the function of the blocks, but often had difficulty understanding the signal flow well enough to know where to place each block within the network. Because of this, they struggle with what relationship the blocks needed to have to each other so they could hear the results they desired. One child, for example, wanted a delay in the sound so she hooked up *Dorothy Delay's Den* by attaching to wherever it most conveniently fit in the network. She did not think to trace the signal flow; this was a common error.

Besides ages, however, there seemed no clear predictor as to which youth would understand the network, to what degree, and how long it would take them to understand it. Whether the child expressed interest in computers, mathematics, arts or sports provided no clear correlation with understanding.

Because a group of children can concurrently interact with the blocks in different ways and on different levels, groups of children successfully played with the blocks even when not every child demonstrated complete understanding of the network behavior. Caitlin and Nolan, a brother and sister, illustrated this in their interaction with the blocks. Caitlin is a 13-year-old "artistic" girl who quickly developed a perfect understanding of how the blocks interacted with each other. Nolan, her 9-year-old brother, loved the blocks too, but had less understanding of the network structure. When playing with the blocks, Caitlin would often build the network while Nolan would explore the sounds that Caitlin created. This division of interactivity within the group was common.

8-10 year olds typically loved the sounds and loved playing with the blocks, but their ability to understand what was happening seemed very limited. If the group were comprised only of this age group, they enjoyed the blocks the most when I showed them simple configurations and gave them only a few blocks to play with. Then they might experiment with limited variations of these configurations. In this context they enjoyed the blocks immensely.

7.1.4.3 Experimentation

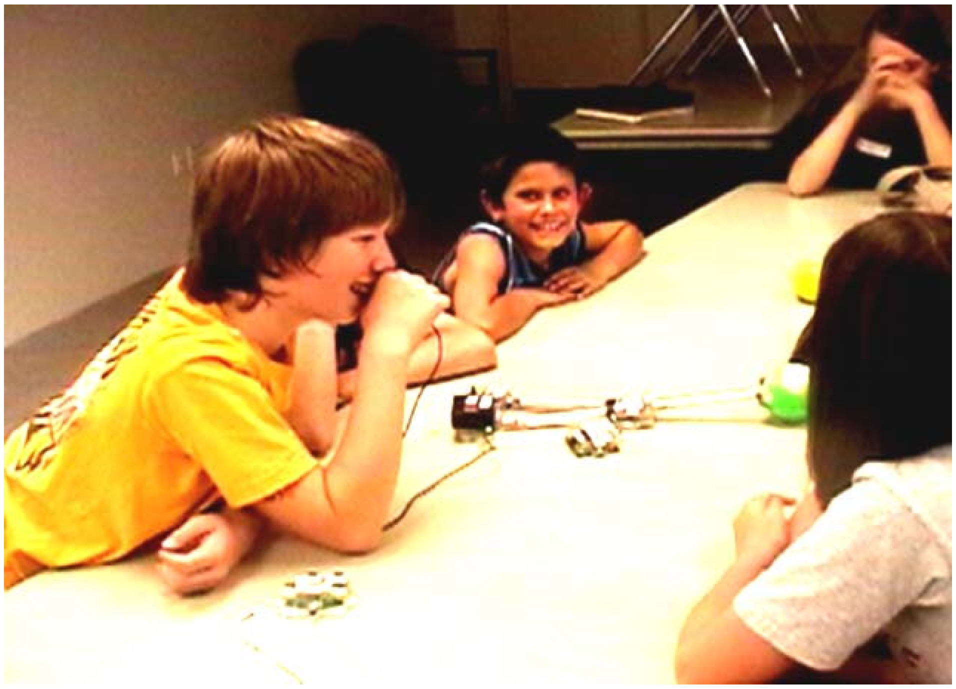


Figure 45: Making sound with *SoundBlocks*

When creating the initial prototypes of the blocks, I had many graduate students and faculty at the lab play with them. These adults seemed hesitant to explore the network before they could completely grasp exactly how *SoundBlocks* functioned. Unless they could predict behavior of the network at all times, they seemed almost embarrassed to explore the network. I wondered at times if some of them feared they would look silly if they started to make connections that might be perceived as nonsensical or foolish.

In contrast to this, the youth of all ages seemed to enjoy playing with the network even when they had little or no understanding of the behavior of the blocks. Instead of being concerned with nonsensical connections, they often enjoyed the surprises in the resulting sounds they created and heard. It seemed as though the kids started their explorations without expectations of what a particular combination of blocks would sound like, and in general they found all of the sounds funny and appealing in some way. They often made mistakes, such as trying to connect outputs to outputs, inputs to inputs, or connecting cables to each other, and they seemed to have less embarrassment then adults when they made these mistakes.

Kids also seemed to enjoy using the microphone spontaneously. Often they would not know what to say into the microphone, so they would make goofy sounds or just talk. In contrast, adults who had tried the environment during the development stage and the parents and teachers who explored the environment throughout development and testing seemed hesitant to make any noises in the microphone at all. Also, some kids seemed to be less inhibited when adults and especially parents were not perceived as actively watching.

7.1.4.4 Social makeup of the youth

The children's impressions of *SoundBlocks* and their willingness to play with it were related to the social makeup of the group. Specifically, youth who were friends were much more likely as a group to become excited and experimental with the blocks. These friendships within the group suggest that these children already share meaningful activities together, and this helps to create a common experience from which they would experiment with the new environment. For example, one group of kids were *Harry Potter* fans. They enjoyed quoting various sections of the books. For these kids, manipulating their voices with the blocks to sound like the various characters of the books proved endlessly entertaining. Another group of kids really liked blues music. One of them used *The Sample Maker* to record himself playing a simple blues melody on a keyboard. Then two of them manipulated their voices using the recorded sample and *The Robot Combiner Diner*.

If the social network within the group was not balanced, a child might feel alienated from the activities and therefore not wish to play with the blocks. Skyler, a 9 year old boy grouped with three 12-14 year-old girls, enjoyed the blocks only for a short time. After perhaps 20 minutes, he asked to be excused. There could be a variety of reasons for this. Perhaps, however, it was simply that he was younger than the other kids, had different interests, and belongs in a different social network for the evaluation.

7.1.4.5 Aesthetics and Shape

Youth showed a lot of interest in the shapes of the blocks. In general, they seemed to like the simpler round primary shapes better than the more complex transparent block shapes. They enjoyed building structures with the blocks and in general were interested in making structures which appealed to them aesthetically as well as aurally.

For example, one child on seeing *SoundBlocks* immediately wanted to hook up as many blocks as possible. He was initially more interested in making sophisticated network relationships than hearing the resulting sounds. He expected that the sounds would be interesting only if the network structure were sophisticated.

Children ages 9-11 especially seemed initially concerned with building structures that they thought were interesting to look at. They enjoyed considering aesthetics, deciding which block would "look right" when placed next to another particular block, or choosing the connector with the correct length. As they tried to make interesting shapes or objects, they would listen to what they created.

In general, the labels on the blocks are not as effective in communicating the behavior of the blocks as physical cues. For example, kids often stubbornly try to force the connectors to go the wrong way. They aren't reading what the blocks' labels for inputs and outputs. If the blocks and connectors offered more visual cues as to how they fit together, the children would likely find the environment more intuitive.

7.1.4.6 The Sample Maker

The Sample Maker was very popular. Students thought the jokes on it (this is not a bomb…or a tomato) were funny. They liked all of the buttons on it and liked being surprised by the random sounds in it.

However, the children also found *The Sample Maker's* user interface confusing. For example, they expected its buttons, especially the "play" button, to work all of the time. It was not intuitive for them that the sampler maker had both a "record/review" mode and a "normal" mode for triggering the samples in the network. Moreover, children would often get confused where on the list a particular sample was stored so they found it clumsy and frustrating to find the samples they had recorded.

7.1.4.7 Audio Distinctions

Children without musical training often had difficulties hearing what sometimes appeared to be obvious differences in the sound. For example, many kids could not distinguish what was changing as they turned the knob on a *Pitch 'R Number* block connected to a *Wild 'N Random Pitch 'R Number* block, as shown in Figure 46.

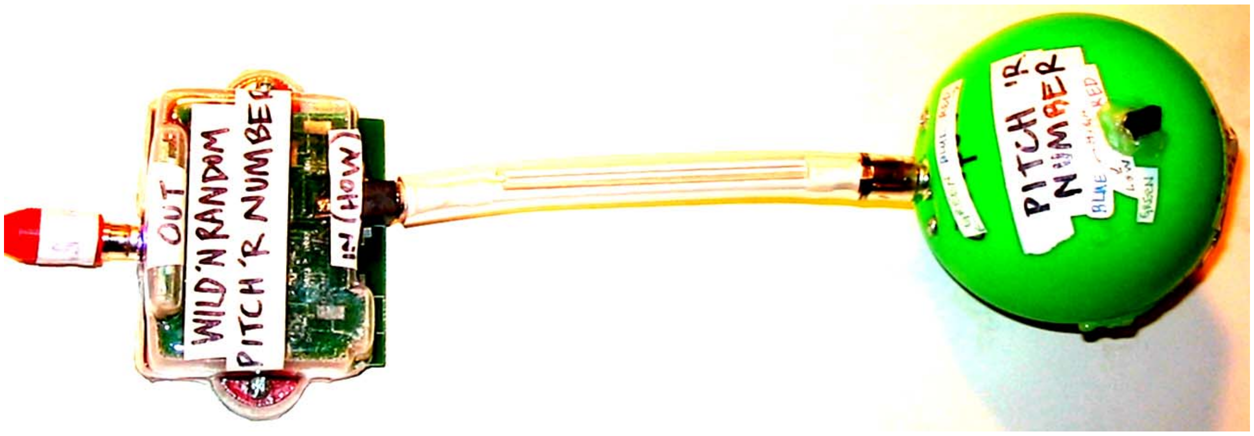


Figure 46: Random pitches change with steady speed

To musically-trained adults, the sound difference seemed obvious: a steady stream of random pitches varied in speed. A similar fairly simple configuration also confused children. Two *Wild 'N Random Pitch 'R Number* blocks connected to each other will generate a stream of random pitches at random speeds, as shown in Figure 47. The children had trouble distinguishing the difference between this configuration and the *Pitch 'R Number* block connected to a *Wild 'N Random Pitch 'R Number* configuration of Figure 46.

Figure 47: Random pitches change with random speed

Many children also did not seem to comprehend the distinction between playback speed (where both the pitch and speed are effected) and pitch shifting (where the pitch is effected but the speed remains constant.)

As the children played with the blocks, their abilities to distinguish between these different aspects in the sound seemed to improve. They seemed to enjoy discovering these distinctions, and sometimes would then be inspired to create networks that brought out changes in the distinctions they had discovered.

7.1.4.8 Youth find even simple manipulations of the sounds around them fascinating

Just about all of the kids who tried *SoundBlocks* were instantly fascinated by hearing their own voices and controlling how their voices were distorted. In general, young kids were quick to grab and shove the blocks between each other. Overall, they were much more fascinated with the sounds themselves and their ability to manipulate them with knobs than with how the network was constructed to create the sounds.

Whenever kids used *SoundBlocks*, there would be a lot of laughing. They seem to find the sounds from the blocks to be very surprising and funny. They also seemed to enjoy trying to surprise each other and themselves with the resulting sounds.

7.1.4.9 Visual cues help understanding

Some children seemed to understand the behavior of the blocks mostly through these blocks' LEDs. These children could quickly grasp the mapping of each block's state to the color of a LED, and then would predict the behavior of the system by describing the LED color. One 9-year-old boy, for example, would describe the network with statements like "the block is blue so we should hear..." Sometimes, because of his angle to the blocks, he couldn't even see the LED colors. After he made these statements, he would sometimes turn the blocks around to check if his predicted LED colors were correct. Apparently, he would imagine what the LED colors might be, then would describe this visual state and predict the resulting sound.

7.1.4.10 Adaptive Properties

Because the blocks have some adaptive properties, the network often produces interesting sounds even when configured in unconventional ways. This was intentional, but it does have a controversial byproduct. Sometimes a group of children would have an idea of how they wanted to manipulate sound, but they would construct the network incorrectly to create this. Although they would get a different result than the one they intended, the adaptive properties of the blocks meant that they might have still created something that they found amusing. They often became districted by the new manipulation and would forget about the original problem they were trying to solve.

There are two ways to interpret the above scenario. In one sense, the adaptive properties of the blocks actually obstructed understanding, since the unexpected results were appealing enough to distract from the original problem. On the other hand, many kids initially showed little patience for the system, and if they had gotten poor results instead of unexpected and yet interesting results, they may have put the blocks away and lost interest.

7.1.4.11 Boredom

When children played with the blocks in several sessions or over an extended session, they seemed to learn about the environment in phases. These phases were divided by what could be described as plateaus, which I initially perceived as boredom. For example, on one occasion two 13-year-old boys at one point seemed to run out of ideas of what to do with the blocks. They appeared to be losing interest. I was tempted to interfere and try to show them "something cool" which I hoped would keep their interest. Instead, I let the scenario play itself out, imagining myself packing up the blocks within the next few minutes when they drifted to some other activity. Instead, this perceived boredom made them wonder about some of the other blocks that they had not yet used or learned about. They picked up the "ask me" block and began initially plugging blocks that they already understood into this block. They heard explanations of the blocks, and it seemed as though they enjoyed hearing about something that they already understood. Then they moved to blocks that they did not yet know about. They then tried playing with these blocks. If they had not gone through this period of apparent boredom, they may not have branched out to try these as yet untried blocks.

7.1.4.12 Kids leading the teaching

If a child within the group was already familiar with *SoundBlocks,* I typically had this child lead the introductory tutorial. This approach had mixed results. Sometimes I overestimated the understanding that the child familiar with *SoundBlocks* had. In this situation, the child would do his best to show network configurations he could remember, but was unable to explain why the networks did what they did or how to troubleshoot when something went wrong. In one case, I had to interfere to help the group understand the blocks. In another case, the child and her friends together puzzled out how they worked. They quickly grasped what blocks would be necessary for the constructions they wanted to create, but it was more challenging for them to figure out the network flow necessary to make these blocks behave as they wished.

7.1.4.13 Expressivity and Real-time Control

Some children were initially very shy of the blocks, but when they heard the various sounds, they opened up. These kids typically found expressiveness by manipulating the knobs for kids who were less shy and made noises into the microphone.

I had expected kids to love the *Wild 'N Random Pitch 'R Number* block. Instead, they took to the standard *Pitch 'R Number* block with the knob. This may be because *Pitch 'R Number* gave them immediately real-time control over what they were doing. They preferred this to the random automation of *Wild 'N Random*, which actually gave them no control over their own product.

7.1.4.14 Other observations

Adults and youth over 15 appeared to enjoy *SoundBlocks* as much as the 10-15 year-olds for which the environment was intended. For example, when showing the blocks to children at a youth camp, the young adult counselors seemed as interested in the blocks as the children. One time, a counselor actually took the blocks away from the kids and started playing with them herself. I was not sure if this was a good thing, but the kids seemed okay with it. I think they enjoyed the counselor taking an active interest in their activities.

In general, children seemed to enjoy opportunities to explore *SoundBlocks* in more than one session. When exploring it a second time, they often recreated structures that they had explored previously. When they were showing the environment to other kids, they often used these recreated structures as a point of departure for explanations.

One music educator felt the environment would be richer if sounds could be layered. This could be implemented if a certain timeframe of sound could be repeated over and over as a loop. Then the user could layer sounds in this timeframe on each pass. As *The Sample Maker* is redesigned, there may be a possibility to integrate this idea. A few children wished to process sound manipulations multiple times by creating a loop within the network. Also, one group wished to record sound into *The Sample Maker*, play this sound back, manipulate it, then rerecord it again into *The Sampler Maker*. This might be possible if loops were allowed or if the system included more than one *Sample Maker*.

I asked all of the kids what their favorite block was. Responses were numerous. However, a common answer was the *Robot Combiner Diner*. I speculate that there are several reasons for this. For starters, the name is appealing. For many of the younger kids especially, when initially hearing the name of this block they would be immediately excited and would repeat the name over and over, apparently just because they liked saying and hearing its name. More than this, however, many of the kids observed that the *Robot Combiner Diner* added character to their voices. This is in contrast to, say, *Polly's PitchShift Parlor,* which alters the pitch of the voice but not the overall character.

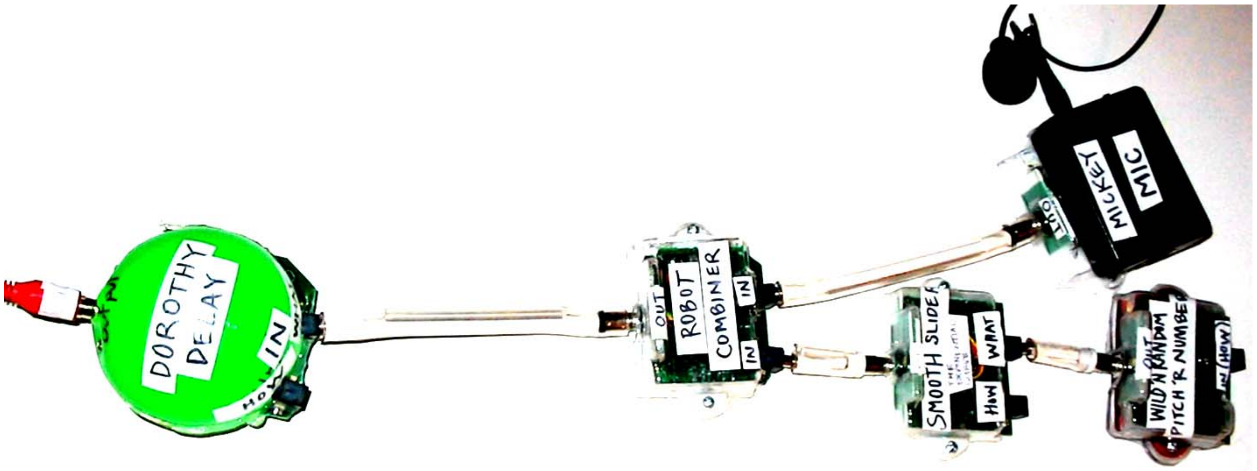


Figure 48: An expressive network

Some others liked the *Smooth Slider*. When asked about this they replied that they enjoyed hearing connections between tones. These connections also can enhance expressivity for some networks, such as the one shown in Figure 48. This configuration changes the inflection of somebody's voice as they speak. Because these changes are unpredictable, sometimes the person's talking takes on surprising meaning or expression, which kids (and adults) found both interesting and entertaining.

Educators observing the kids felt that the kids learned a great deal about the network structure and the signal flow of the network through play with the blocks. One interpretation of what was happening is that the kids were, in a sense, creating a tangible program, which they then debugged with their ears. In fact, their complete understanding of the blocks, these blocks' functions, and the blocks' relationships were all governed by the sounds they heard.

7.2 *SoundScratch*

7.2.1 *Overall Evaluation Strategy*

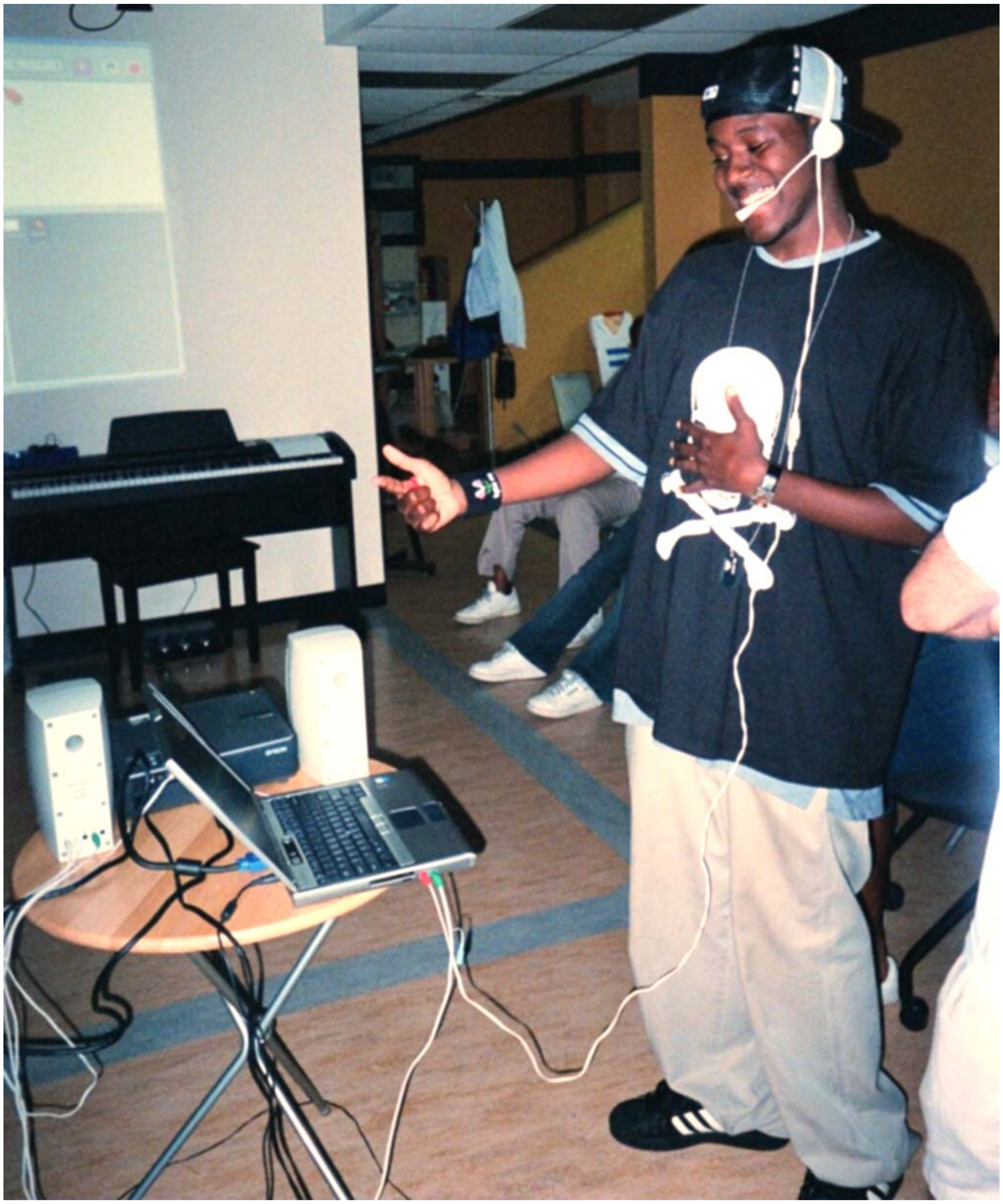


Figure 49: *SoundScratch* at the Sound End Technology Center

SoundScratch was evaluated in a manner similar to *SoundBlocks*. Youth ages 10-15 explored the environment, and music educators and I appraised the usability and expressiveness that the kids seemed to experience in the environment. We also evaluated the youth's experience understanding the sound manipulations themselves. These youth were from a variety of backgrounds. Some were already familiar with *Scratch* or *MicroWorlds Logo* and some had no programming experience whatsoever. Some had advanced musical training and some did not. Inner city youth from underserved communities participated as well as youth from more privileged communities. Evaluations took place at The Sound End Technology Center of Boston, The Computer Clubhouse at the Museum of Science in Boston, and at the New Hampshire Music Festival at Plymouth State College in Plymouth, New Hampshire. Youth at Plymouth were the children of festival participants.

As with *SoundBlocks*, I designed a tutorial for *SoundScratch* to be about 20 minutes in length. The tutorial was divided into six parts, and provided a short introduction to *Scratch* as well as the various possible sound manipulations in *SoundScratch*. During the tutorial, youth manipulated the blocks and used the microphone, as directed by me. By the end of the tutorial, they were free to explore the environment as they wished. Music educators and I observed their exploration and served as a resource for questions if the youth needed any help. We also asked questions and initiated challenging problems if we felt this to be helpful motivationally for the children or for our own evaluation. Our intention was to best understand the youth's perception and understanding of the environment in the short time we had with them.

As well as we could, we also tried to learn a bit about the youth themselves. We asked them questions about what their favorite classes were, whether they liked sports, art, mathematics or English, and what their favorite activities were. Through these questions, we got to better understand how youth with different styles of learning experience the environment.

As the youth worked within the environment, we recorded pictures and video. We also archived some of their creations, although unfortunately some of these have been accidentally lost. We have reviewed the media we still have for a better understanding of the usability and expressiveness the youth experienced with the environment, as well as their understanding of the various sound manipulations.

7.2.2 Tutorial Introducing SoundScratch *to Youth*

Figure 50: Members of the Museum of Science Computer Clubhouse using *SoundScratch*

Based on what had and had not worked in presenting the tutorial of *SoundBlocks* to youth, I developed a tutorial for *SoundScratch*. The tutorial consisted of six parts.

7.2.2.1 The Scratch *Environment*

- *Purpose: Introduce the basic concepts of* Scratch *using its visual elements.*
- Explain that there is a costume we can manipulate: a cat.
- Have youth click on the blocks to move and rotate the cat.
- Demonstrate the script window by having youth drag the "move" and "rotate" blocks into this window.
- Show how the blocks interlock: have a child interlock the "move" and "rotate" blocks together, then double-click on these blocks.

- Demonstrate the "forever" block: have a child insert the "rotate" and "move" blocks into the "forever" block and double-click on it.
- Demonstrate the stop button.
- Connect a "when green flag is clicked" block to the "forever" blocks. Show how this configuration rotates the cat in a circle when the green flag is clicked. Stop it with the stop button.
- Show that we can replace the cat with our own picture.

7.2.2.2 Introducing Sound Manipulations

- *Purpose: Demonstrate simple sound manipulations in* SoundScratch.
- Show the "sound" category to *Scratch* and the corresponding blocks.
- Have a kid demonstrate the "set sound to" and "start sound" blocks.
- Have a kid change the pull-down menu in the "set sound to" block from "pop" to "meow".
- Show pitch shifting: have a kid create a script that sets the sound to "meow", sets the pitch, then starts the sound. The kid chooses the pitch by typing a number into the "set pitch to" block.
- Have a child replace the "set pitch to" block with a "set tempo to" block. As they type numbers, the meow plays at different speeds, as well as backwards and forwards.

7.2.2.3 Recording

- *Purpose: Introduce basic possibilities for youth to create their own personal sounds. Introduce sound effects.*
- Show the "Sounds" category (where "pop" and "meow" are listed.) Explain that any .aiff file can be manipulated, just as we already have with "pop" and "meow."
- Have a kid record a sound and name it.
- Manipulate the new sound as in 7.2.2.2.
- Demonstrate the sound effects by replacing the "set pitch to" block or "set tempo to" block with a "set sound effect to" block. Experiment with various sound effects within this block. Ask the children to describe what each one sounds like.

7.2.2.4 Loops and mouse manipulations

- *Purpose: show how sound manipulations can be changing continually, and can be mapped to mouse movements.*
- Ask the youth if they can remember the "forever" block introduced in section 7.2.2.1. Explain how this block can function with sound as well as with visual costumes.
- Introduce the "resume sound" block: have a child place this block inside a forever block and double-click. They will hear the selected sound repeat from beginning to end continuously.
- Ask the youth what might happen if we insert a "set pitch to" block inside the "forever" block as well. Have them try it. Once the block is inserted, they can again type in numbers directly into this block and hear the resulting change immediately.
- Show youth the "mouse x" block in the "sensing" category. Show how this value is continually changing as we move the mouse.
- Ask the youth what would happen if we replace the number in the "set pitch to" block with the "mouse x" block. Have a child try it.
- Ask the youth what other effects they wish to manipulate with mouse movements and have them create the necessary configurations to try these things.

7.2.2.5 Using live microphone

- *Purpose: demonstrate that live sound from the microphone can be manipulated as easily as recorded sound.*
- Show how the "set sound to" block has an option for "live microphone." Have a child choose this option, then "start sound" to hear her own voice.
- Introduce the "delay" block: have a child double-click this block with a setting of 500 (.5 seconds).
- Introduce the "reset sound effects" block: have a child double-click on this block to remove the delay.
- Have youth alter the manipulations described by the set of blocks constructed in section 7.2.2.4 by changing the "set sound to" block to "live microphone."

7.2.2.6 Sprite Independence

- *Purpose: Show how multiple sounds can be manipulated independently and simultaneously.*
- Show how there is a list of sprites in the bottom right hand corner, and that currently we have two sprites: the active sprite and a background sprite.
- Explain the concept of sprites.
- Have a child create a simple sound manipulation so that "live microphone" is pitch-shifted continuously with "mouse x" when the green flag is clicked.
- Have a child clone this sprite twice, so now there are three copies of this.
- A child now clicks on the green flag and hears the microphone output, perhaps a bit louder and with some mild chorusing artifact. Ask the youth what is happening. Help them deduce that all three scripts are running independently and simultaneously.
- Show how we can edit the script of each sprite independently: have a child alter one sprite by getting rid of one of the pitch shifting blocks, then alter another sprite by having the pitch manipulated by mouse y. We now have the set of scripts described in Figure 20.
- After the children play with this, show how set values for the pitch shift can create various chords.

The described tutorial was sometimes abbreviated, depending on the age of the youth and their demonstrated understanding of the environment during the tutorial. At the end of the tutorial, the children were free to play with the environment, guided by their own imaginations. While the music educators and I continued to observe, we kept our interference at a minimum. The free play time period typically had no specified time window. It was as long as the youth wished and were available to continue to create within the environment.

7.2.3 Tutorial Presentation

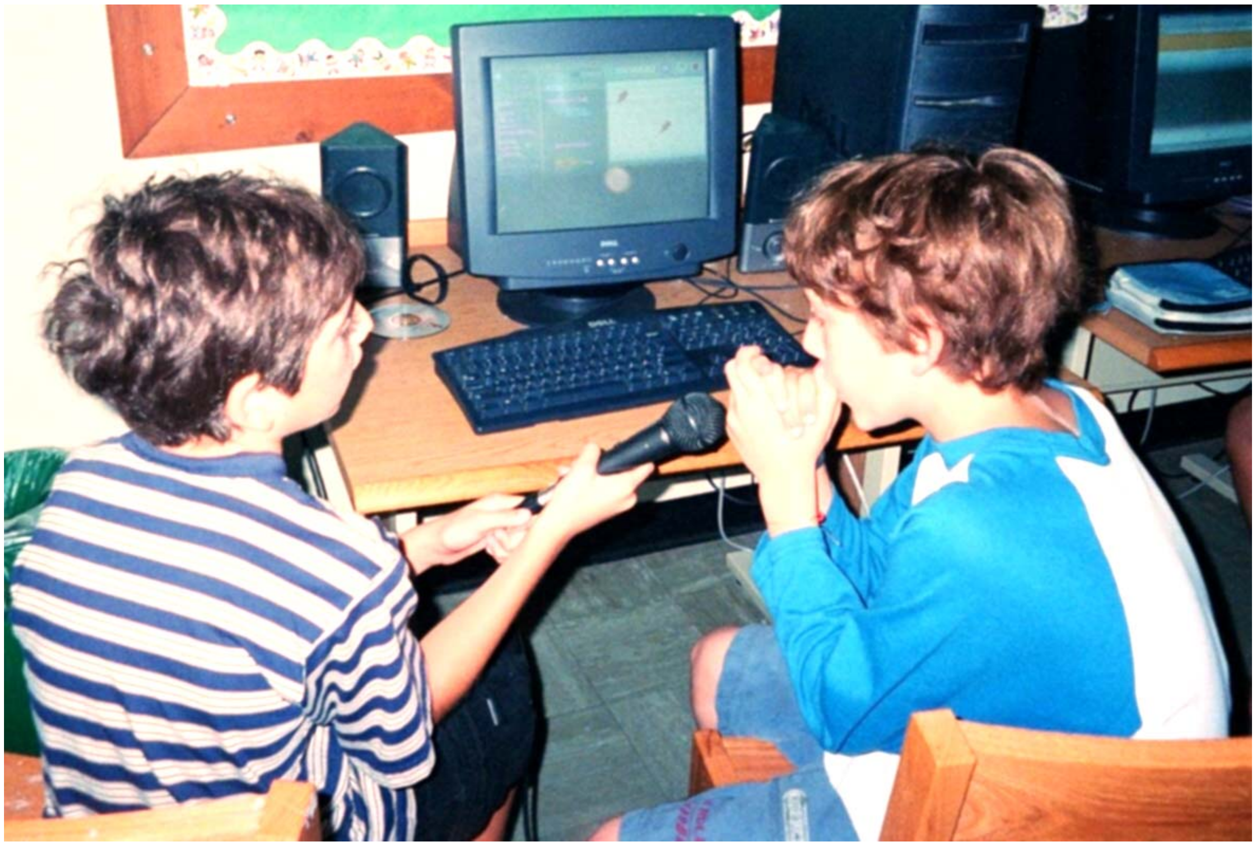


Figure 51: Youth in New Hampshire with *SoundScratch*

SoundScratch, like *SoundBlocks,* is intended to be an environment which children are free to explore as they wish. However, unlike *SoundBlocks, SoundScratch* does not require a special setup and can be quickly running in most typical computer rooms. Therefore, the tutorial for *SoundScratch* was typically more informal than that for *SoundBlocks*.

SoundScratch tutorials were always conducted at a center with multiple computers and where kids were currently using the computers, typically for games. Usually, I would sit down at a computer, get *SoundScratch* running, then ask the youth in the center if they wanted to "check it out." If I got no response, I just started playing with it myself. Invariably at this point, some youth would begin to notice the manipulated sounds I was creating and would become interested. In this way, they chose to explore *SoundScratch* only if they wished and on their own terms.

There was one notable exception to this approach: at *The South End Technology Center* in Boston, I conducted the tutorial as a formal presentation. The scenario was as follows: *The Future of*

Learning group runs a day camp at *The South End Technology Center,* which ends at 3PM. Usually at 2:30 they have a roundtable discussion, before the youth can leave for the day. On July 19, 2005, I used their 2:30-3:00 time to introduce *SoundScratch* to them instead. We invited any youth who were available after 3PM and wished to explore the environment further to work with me individually after the camp ended that day.

7.3 Observations

7.3.1 Live Microphone is especially attractive

In general, youth enjoy hearing the results of sound manipulation scripts in *SoundScratch.* They especially enjoy taking turns making sounds into the microphone and hearing the resulting real-time manipulation of their voice. They find hearing themselves and hearing their friends' voices manipulated in this way very funny.

Younger children especially find simple manipulations with the microphone endlessly fascinating and a complete world to explore in itself. Tyler, for example, was a 10-year-old who loved to hear his voice pitch-shifted. He spent a surprising amount of time just talking into the microphone and hearing his voice pitch-shifted to be one octave higher. He pretended during this time that he was a mouse. Eventually, I showed him that with a "forever-set pitch to mouse x" script he could change the pitch as he moved the mouse. Using this configuration was interesting enough for him that he had no desire to explore further possibilities of the environment for quite some time. He started making up a story, where he played both characters in the story. For the "big fat" character he pitch-shifted his voice to be low. For the little mouse he pitch-shifted his voice to be high. As he told his story, he changed the pitch shifting parameters just by moving the mouse.

7.3.2 Programming

While it is easy to engage youth with *SoundScratch* when I describe to them how to manipulate the blocks to create interesting scripts, it seems much harder to motivate them to create their own scripts without my lead. The hurdle may be the programming element, which has some learning curve and which they appear to be reluctant to venture into in general.

This hurdle became evident in different ways with different groups of kids. At *The Sound End Technology Center,* kids would wait for me to come up with a new idea for a manipulation then have me describe how to situate the blocks for this manipulation. If I provided all of this guidance, then they enjoyed playing with the resulting sound. However, if I asked them for ideas for sound manipulation or how we might create the script for a given idea, they seemed to lose interest quickly.

During a free-play time period in New Hampshire, one kid was full of creative ideas for how the sounds might be manipulated, but he would not consider how he might create these manipulations himself. Instead, he would immediately ask me to do it for him. I would challenge him to figure it out with me and he was extremely reluctant to do this. Moreover, he did not seem to enjoy this process. The problem-solving element of it apparently did not appeal to him. He seemed much more focused on the results.

When youth were reluctant to create scripts, I often found it helpful to focus on how they might manipulate sound just by clicking on various blocks and changing the parameters within these blocks. For example, a kid might drag out two “set pitch to” blocks, and insert the number “50” into one of them and “200” into another. He might then set the sound to the microphone and click on the “start sound” block. He could now click on either of the “set pitch to” blocks to immediately change the pitch of his voice to an octave higher or an octave lower. While not a full-fledged script per se, it at least offered an introduction to the idea of programming with scripts, and that they could control the manipulations of sound themselves.

7.3.3 Visual Element

When designing *SoundScratch*, I had underestimated how much the visual elements of *Scratch* would contribute to the sound manipulations that youth would find interesting. Many youth, in fact, would start their exploration of *SoundScratch* by first creating an animation, then adding sound to that animation. In these cases, it seemed as though the visual element was actually providing the motivation the youth needed to inspire them to explore the sound manipulations.

From the youth’s perspective, it appears that making the sounds alone did not provide the richness for a complete story or creation. On the other hand, the visual elements seemed to come alive and excite them as they added sound to them. Within this context especially, they found it fascinating to manipulate their own voices to sound like things they don't recognize. Here are some examples:

- Frank was a member of the *Sound End Technology Center*. He began his 1½ hour exploration of *SoundScratch* by writing a script to make the cat move around the screen. He developed this script so that the cat would rotate, and would follow the mouse. At this point, he integrated sound by having a sound start when he pressed the mouse button, then stop when he released the mouse button. He reversed these functions so the sound started when he released the mouse button instead. Then he began to explore the possibilities of mapping pitch and speed with cat position. This presented some interesting challenges of how to get the pitch to drop when the cat went down and the pitch to go up when the cat went up.
- One member of *The Computer Clubhouse* had worked with *Scratch* previously. She had recently written a story using *Scratch* and wanted to use the sound extensions to get the characters in her story to start talking to each other by manipulating her own voice to represent the different characters.
- Natasha in New Hampshire was initially interested only in the visual elements of the program. We worked together for 1½ hours with me helping her figure out how to make a basketball player dribble a ball and shoot the ball into the hoop. I was really amazed at how quickly she could grasp the language and how she could figure out how to do stuff pretty much all on her own.
- A group of three boys and a girl in New Hampshire were initially interested only in the animation. The girl drew three costumes: Saturn, Earth, and a black background. The boys drew a spaceship costume. Through play, the four of them eventually evolved a plot: the space ship would twirl in space, run into Saturn which would then twirl into space, then run into earth which would twirl off of the screen. After they created the scripts for this animation, they wanted to add sounds. When the space ship hits Saturn, they recorded "oh no!" which they pitch shifted down, added echo, then ran a high-pass filter to clear up the muddiness in the sound. After Saturn says "oh no!" then the space ship says "yes" with a pitch shift upwards of about two octaves and also a high-pass filter. They thought this was really funny. Then Saturn twirls around until it hits Earth. When it hits Earth, Earth says "we're all gonna die!"

which was played without manipulation, and several of them saying it at once. The final product was an impressive achievement representing 2 ½ hours of play.

- Tyler, who had been making up stories about a mouse and a big fat man using the sound manipulations, wanted to draw a mouse to support his story. In this example, the sound manipulations supported exploration of the visual elements of *Scratch* rather than the other way around, as was more typically the case.

7.3.4 High Level Structures and Synthesis

Many users and members of the *Sound End Technology Center* especially were very interested in using *SoundScratch* to create loops and manipulate synthesized sounds. This is consistent with kids interests: many of them listen to rap and hip-hop and they wish to explore this style of music and the synthesized sounds they hear in this music.

High-level structures using synthesized sounds are possible through scripts within *SoundScratch*. However, they are not as easily created as the low-level manipulations for which the environment is intended. Moreover, the sounds themselves must be provided from outside the environment. In the future, *SoundScratch* should include a group of these sounds in its own directory. Also, there should be some examples of high-level loops, which can imitate loops heard in popular music through scripts.

7.3.5 Computer hardware

Installing and running *SoundScratch* in various environments continues to be a challenge. On Windows 2000 machines, the necessary windows script will not complete because of some differences between Windows 2000 and Windows XP. Consequently, *SoundScratch* will run on these machines, but the sound has some minor clicks occasionally, and Csound will not automatically quit when a user quits *Scratch*.

Many computers in public areas are purposely crippled: software cannot be installed on them, scripts cannot be executed, or they present various alarming warnings and offer limited functionality. *SoundScratch* would not run at all in *The Computer Clubhouse*. It ran with a rewritten script in New Hampshire that limited functionality in some minor ways and prevented Csound from quitting. At *The Sound End Technology Center* it ran, although it produced alarming warnings.

There was a surprising amount of variance in how well computers were able to run computationally intensive sound manipulations in *SoundScratch*. Moreover, there seemed to be little correlation with expected performance using these computers. Specifically, the age of the computer or the CPU in the computer did not seem to indicate how well *SoundScratch* would run. Some Pentium 4 computers were unable to process even three sounds at once while older generic machines were able to do this easily. Perhaps this is related to the basic health of the operating systems on these various machines and the extent to which they are crippled.

The most frustrating situation was in New Hampshire. The public computers there periodically delete user files stored on the local hard drives of these computers. I learned about this when the youth's creations during the various sessions were deleted while they were using the environment.

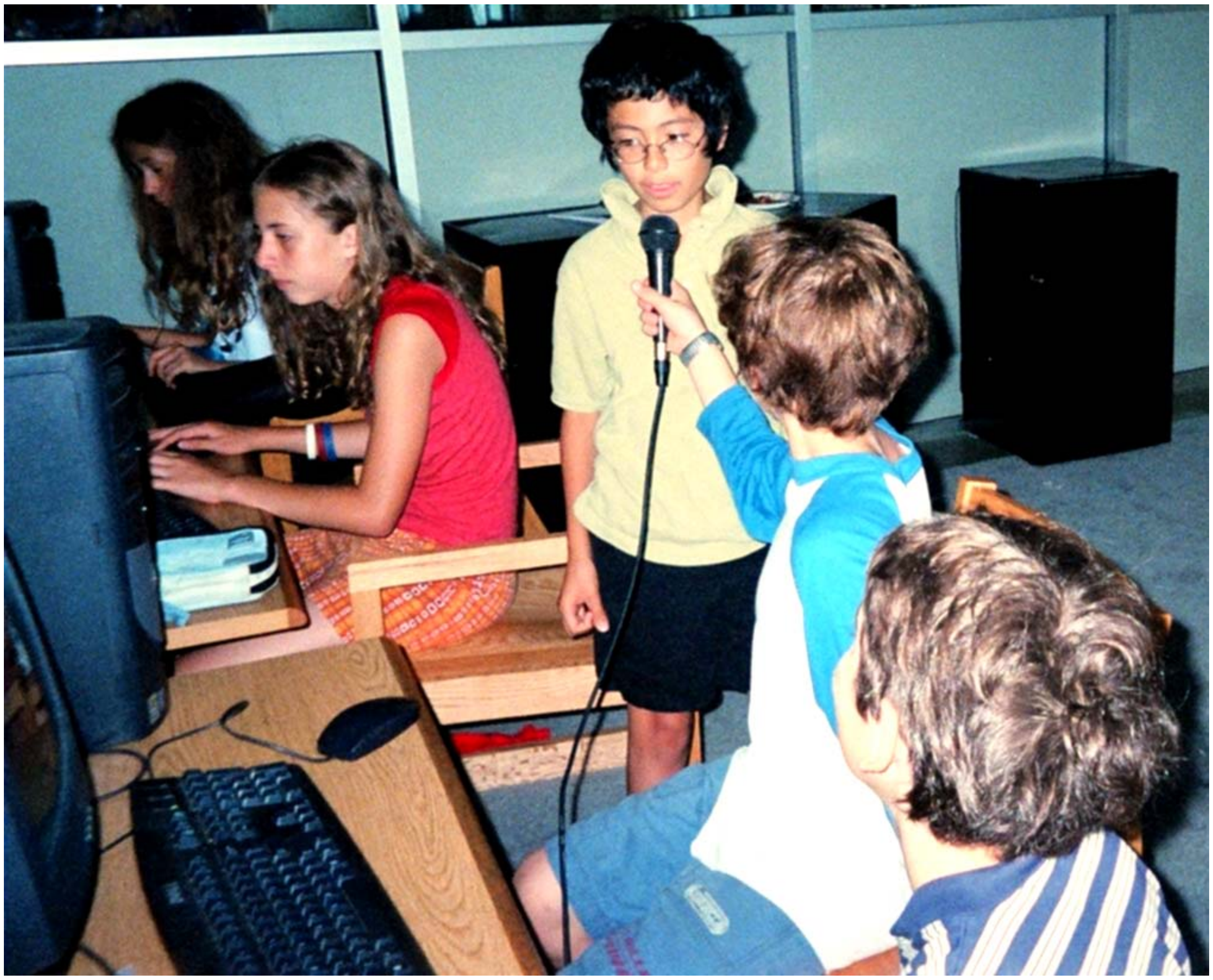


Figure 52: Youth interacting with *SoundScratch*

In general, children find it difficult to interact with each other and with the computer at the same time. Instead, they often end up using different computers. In this way, they can all explore the environment, but they cannot explore it as a team, bouncing off of each other's ideas and learning to work together. For example, during one session in New Hampshire, one child explored sound manipulations with the microphone and other children would come up and try his scripts, but there was no easy way for them as a group to be actively involved with creating the manipulations. Those who were very interested asked if they could use a different computer.

7.3.6 Connecting with other programs and other sounds

Many youth were interested in using *SoundScratch* with other sound software. Some of them could do this with some guidance. This made the environment of sounds that they could explore much richer.

For example, Frank at *The South End Technology Center* generated both a sine wave and white noise using an open source program called *Audacity*. (He did not initially intend to create these waves. They were a natural result of exploring the program itself.) He then imported the resulting sound files into *SoundScratch*, which provoked him to ask some interesting questions: What does "set tempo" mean for a sine wave? What does "set pitch" mean for white noise? He experimented with the blocks to answer these questions, and he developed his already-existing scripts in *SoundScratch* to incorporate the new sounds.

Some youth wished to export the sounds resulting from their sound manipulations into other sound software. It is possible to do this using *Audacity*, but it is not a straightforward process. If a block were added into *SoundScratch* with this capability, this would be much easier to do. Moreover, *SoundScratch* itself could use the newly-recorded sounds for further manipulation.

Many children thought to sing into the microphone Karaoke-style, with their voice altered in some way. If the environment could import .wav and .mp3 files in addition to the .aiff files it can already import, this functionality would be facilitated.

7.3.7 Differences among youth

I found a great deal of variance in how youth perceived *SoundScratch* and how they perceived the tutorial I gave for *SoundScratch*. Youth at *The Computer Clubhouse* were especially passive during the beginning tutorial. It almost felt as if they were watching TV when they were watching me. They eventually warmed up to talking into the microphone as I did the manipulations. I demonstrated what was possible and they found it funny and amusing. After we all took turns playing with it, singing, etc., two members took an active interest.

While the response I get from youth has something to do with the projects themselves, I would speculate that it is also very dependent on their relationship with me. If the kid has a reason to trust me and senses some rapport between us, she is more inclined to be interested in the projects. If she has no reason to extend trust, as might be the case if I clearly come from a different culture or race, she may be willing to offer less trust.

7.3.8 Interference with observers

Throughout interacting with all of the youth, it was often unclear how much to interfere with their own explorations. For example, Natasha, 12 years old, wanted to animate a ball shooting through a hoop. She didn't know how to do it and asked for help. I didn't think it appropriate to start talking about parabolic motion, acceleration and velocity, etc. In the end, I did give her a cursory explanation and had her write the script. I feel mixed about this. On the one hand, she asked and why should I deny her knowledge? On the other hand, I think she may have come up with her own solutions. While her solutions might not have looked as good or have been technically correct, it may have been more creative for her to devise these.

One boy at *The Computer Clubhouse* seemed to get frustrated easily. When things did not work, his solution was to delete entire projects, then start from scratch. It was hard to judge how much to interfere with his process so that he might not get so frustrated.

7.3.9 Other observations

Children who used *SoundScratch* for more than one session in general showed a surprising amount of retention. They remembered how the various blocks functioned and how to navigate through the *Scratch* environment to get the results they wanted. This suggests that they find reasonable usability and understanding in the environment.

When presenting *SoundScratch*, I had been fearful that kids would not be interested. I worried that the flashy graphics common in all commercial computer games would overshadow the possibilities for creation in *SoundScratch.* I was wrong. Kids show an interest in telling stories through animation and sound using *Scratch.* Even though their animations are simple relative to commercial games, they are proud of them and show them off to adults and other children. The children tend to be very supportive of each other.

8. Conclusions and Future Work

8.1 SoundBlocks *and* SoundScratch evaluation

SoundBlocks and *SoundScratch* were created as a way to explore if sound manipulation might offer the same potential for learning and for expression as image manipulation. As an initial venture in this arena, we have explored what it means to offer sound manipulation within the Constructionist framework. Can sound be programmed like images in a programming language for youth? Can a tangible environment for sound manipulation offer a way to learn about networks and data manipulation within the framework of exploring sound?

Both *SoundBlocks* and *SoundScratch* differ from earlier Constructionist approaches to sound in that they focus on manipulation of sound generated from the youth and the environments around them. This contrasts with the more typical high-level approach where users are given notes as the smallest unit they can manipulate, all notes are synthesized and cannot be changed within the system, and the notes can relate to each other only chromatically. The choice to try an alternative approach was intentional. It is part of a more general query exploring whether youth might find more meaning in sound manipulatives if these manipulatives are applied to sounds unique to them and their environment.

Quantitative evaluation of environments like *SoundBlocks* and *SoundScratch* is difficult. However, through my own and music teachers' observations, there seems to be some consensus that students did find enjoyment in exploring both systems. Specifically, just about all of the youth, including those who claimed to be unmusical or hate musical instruments, seemed to find great enjoyment in making sounds, and then in seeing how these could be manipulated within the environments. The sound generated from both systems was of high quality, and this may have contributed to how the students reacted. It makes the sounds seem interesting and personal, as they can hear their own inflections and recognize various qualities in their voices and the voices of their friends.

Some observers (including me) seemed surprised at what distinctions the youth did not seem to notice when manipulating the sounds. As they explored the system, some seemed to gain some awareness of these distinctions. The combination between understanding the configurations they made (code or network) and the visual feedback (computer screen or LED) with the sound seemed to make a mutually supportive combination in developing these distinctions.

8.2 Comparing the Two Environments

Although *SoundBlocks* and *SoundScratch* both explore digital sound manipulation within the context of Constructionism, this is where the similarity ends. Therefore, I suspect that a direct head-to-head comparison between them would have produced meaningless results. However, it is interesting to reflect on how the differences between the two environments elicited different responses from users.

Youth are very used to flashy and exciting manipulations of video and audio on computers. When presenting to them a computer environment such as *Scratch*, the comparison with the latest X-box or Sony Playstation game may be inevitable. There is no way for any children's programming language to reasonably compete with an X-box game, so on some level it can be difficult to initially generate enthusiasm from the youth. In comparison, *SoundBlocks* seems very removed from the computer, so youths' expectations appear to be different. The youth have nothing to compare it to. They have never seen the blocks do anything before, unlike the Dell Desktop, which might have been running *Civilization* 5 minutes earlier. Thus, the youth appear to be more inclined to play with *SoundBlocks*, interact with *SoundBlocks*, touch *SoundBlocks*, feel *SoundBlocks*. They seem a lot more forgiving of bugs within the system and overall appear to be more easily impressed. This makes me wonder if perhaps a tangible environment promises a more fun, interactive and powerful learning environment for youth within the domain of sound.

Both projects addressed the problematic issue of data manipulation. The problem is that digital sound manipulation is usually described as a relational network of various types of data, as described in 5.8.1. *SoundBlocks* addressed the problem by adding adaptive behavior to blur the distinctions between various types of data. While this solution has its benefits, the problems of how to define the adaptive behavior become increasingly difficult as the system scales to a larger project with bigger networks and more varied blocks. Also, some might argue that users will have trouble understanding the definitions of the blocks, since the blocks change definitions in various situations. This issue did not arise in these evaluations with the current set of blocks.

SoundScratch addressed the problem by hardcoding the relational data network into the system so that it would be a black box to the user. Some observers of the system who are familiar with digital audio processing were critical of this choice and felt that perhaps it took much of the interest of exploring the sounds away as the youth couldn't explore various arrangements of effects.

8.3 Further Possible Developments of SoundBlocks

The *SoundBlocks* environment shows a lot of promise. Already the environment is rich enough that youth seem to enjoy exploring it and perhaps find some personal expression within it. We intend to continue to develop the system. Here are some of the expansions we are looking at.

8.3.1 Portable

SoundBlocks should be completely portable. Users should be able to carry it with them wherever they go, and perhaps hold it in their hand. They should be able to build truly three-dimensional structures. To this end, we have already redesigned the circuit board to fit in the size of a golf ball. We are looking at changing to smaller connectors that retain the robustness of our current RCA and DC jacks. We are also exploring how to move the synthesis engine to a DSP chip, a goal since the genesis of the project. This switch will happen in two phases. In our first phase, we are porting the Python code to run in C on an Atmel Mega 8535. In the second phase, we will explore possibilities for integrating this chip to a DSP sound synthesis engine.

8.3.2 Support more varied structures with true three-dimensional support

We are exploring connectors that are much smaller than the current connectors and aren't mounted to the circuit board. This will allow us to create a true 3-dimensional building platform. Also, blocks should be able to plug into each other directly and connectors should be able to plug into each other. We are also looking at using magnets as connectors, as has been done in similar projects (Gorbet, Orth et al. 1998; Newton-Dunn, Najano et al. 2003). Our one-wire protocol makes the magnet option attractive, as we do not have to worry about alignment issues that arise when more then two connections are necessary.

8.3.3 Redefine the Sample Maker

The Sample Maker needs to be completely independent from the rest of the system. Specifically, it needs to be small, self-powered and easier to use. Users should be able to perhaps attach the sample maker to their key chain so they can easily record what interests them as they go about their daily lives. We already have initial sketches for a Sample Maker that runs off on Atmel ATTINY microcontroller, uses a simple DAC, and uses Compact Flash.

As an alternative, we are considering eliminating the hardware of the Sample Maker entirely and instead designing software for cell phones that will let these devices assume the role of Sample Maker.

8.3.4 Add support for multiple audio streams

Currently, *SoundBlocks* supports one speaker and one microphone within the network. If the system could support multiple concurrent audio, perhaps it would be possible for multiple microphones and multiple speakers to be producing simultaneous independent output. Such a system would increase the possibilities for interactivity. It is easy to imagine games and stories youth could make up together where sound is passed from one place to another, and the youth can hear the sound moving from one speaker to another as it passes through.

8.3.5 Expand visual feedback

An RGB LED gives some information, but perhaps for a large-scale network it is not enough. We are looking at designing a block that has a multi-character LCD readout that can be attached between blocks. The LCD block will be transparent to the network function, but the block itself will provide detailed information about the signals being passed through it. This could be a useful debugging tool.

8.3.6 Have varied shapes for the blocks that symbolize their function

If the blocks looked like the functions they represent, they may be more intuitive to users. How should a pitch shifter visually look different than a vocoder? What might a volume block look like to distinguish its audio function? When answering these questions, we wish to remain especially aware of the importance of the aesthetic appeal of the blocks. The blocks need to appeal to the users visually. This has been clear throughout design and evaluation of the project, and its importance cannot be overstressed.

8.3.7 Add more blocks which alter the inherent character of the sound

Two very popular blocks among users were *The Robotic Combiner Diner* and *The Smooth Slider*. These blocks were more popular than *Polly's PitchShift Parlor*. At first it was unclear why this might be the case. When we asked users about this, they told us that both *The Robotic Combiner Diner* and *The Smooth Slider* do more than just change the sound. The operations of both of these blocks alter inflection and expression of people's voices. Therefore, we are looking at expanding the system to include more blocks that, like *The Robotic Combiner Diner* and *The Smooth Slider*, change the inherent character of a sound.

8.3.8 Make **SoundBlocks** *compatible with other audio environments*

One serious criticism which music teachers made of the system was that users could not conveniently take the sounds that they had made home with them. *SoundBlocks* offers no way to provide "refrigerator audio art." Users should also be able to easily import and export audio out of the system.

8.3.9 Add support for higher level structures

While we wish to make manipulating the sound the principal focus of the system, we also wish to add support for large-scale structures. Users should be able to define sections of sound that they can trigger with MIDI. To address this concern, we have already designed a MIDI block. The MIDI block can accept MIDI data from any standard MIDI device and send it into the network.

The system also needs some sort of sequencer, and we are having ongoing discussions of what exactly this would be. As has been the goal with the entire system, we want to make the sequencer both intuitive and inexpensive. This may spill into a rethinking of the basic function of the sampler or how it is used.

8.3.10 Send audio through the network

The current version of *SoundBlocks* appears to send audio through the network. Internally, however, audio data is never transmitted in the network. Because of this, *Micky Mic* transmits audio wirelessly. There are several disadvantages to this approach: cost, expandability to support multiple streams of audio, and power. Therefore, we are looking at ways of adding high-speed to our one-wire protocol so that we can eliminate the wireless connection.

8.3.11 Add sensors

The original concept of *SoundBlocks* was as a new musical instrument with an adaptive interface. This requires that some blocks must be made sensitive to gesture motion of various types. To accomplish this, we are looking at various inertial measurement unit sensors, as well as light, sound, and proximity detectors. We also are experimenting and very excited about a stretchable fabric we have found whose resistance changes as it is stretched. We imagine using this fabric, for example, to create a delay block which the user actually stretches to change the delay.

8.3.12 Add support for loops and multiple outputs

Networks can be especially exciting to explore if there is feedback allowed within the system. Currently, our system allows no feedback. We are looking at what would be involved in adding support for feedback and, a related issue, multiple outputs. This is not a trivial task. Even some virtual environments, such as Pd, do not allow loops in their networks. Also, there are some concerns with the one-wire protocol supporting various combinations of loops and parallel signals.

8.3.13 Clone blocks, users write code

A clone block would be a block that can take on the function of a network of blocks. How this would be done and how it would be represented are still under discussion. What is clear is that it would be powerful if users could setup a network configuration and then somehow assign one block, the clone block, to assume the functions of the entire network. It could be that the user would attach the network's inputs to the clone block's inputs and the network's outputs to the clone block's outputs. It could also be that the LCD block mentioned in section 8.3.5 will identify the clone block's network configuration when plugged into the clone block. A clone block can be thought of as a macro.

Since the internal code is very modular and all of the virtual patchbay and instance manipulation within Csound is wrapped up in macros, it is already easy for users who know Csound to add another block to the system. Extending this idea, it is imaginable that users themselves could assign code to the blocks. This could be along the lines of how Crickets are programmed, except using a Csound/LOGO-like language interface.

8.3.14 Create GUI and release as open source

The structure of the code, as explained in section 5.2, is such that the network discovery is completely separate from the rest of the code. Therefore, it would be relatively easy to create a completely separate module that replaces the network discovery implementation. Specifically, we have already started to design a GUI that could be used to manipulate blocks on screen instead of the physical blocks. The GUI would be similar in some ways to Max/MSP and Pd, except that the blocks would be programmed in Csound, have adaptive behavior, and each would have color attributes as well sound attributes. If we were to complete the GUI and release the entire code as open source, users could extend the environment with their own Csound and Python code to add blocks and extend adaptive behavior.

Furthermore, it may be possible to integrate both the GUI environment and the physical environment together, building on the ideas of the first iteration's use of masses and springs.

8.3.15 Connect **SoundBlocks** *over the Internet*

We are considering ways in which users might be able to have their *SoundBlocks* communicate and interact with each other over the Internet. It might be possible for people's environments to affect each other, thus creating a larger network of blocks over the Internet, each comprised of the smaller local blocks.

8.4 Further Possible Developments of SoundScratch

8.4.1 Look at Timing Issues

When writing a script such as a simple metronome, users notice irregularities in the timing of the sounds. It is unclear right now why that is. With sound it is absolutely crucial that the timing is perfect. This problem needs to be investigated further.

8.4.2 Add blocks

Currently *SoundScratch* supports absolute settings as in "set filter to x" or "jump to x seconds." Relative positions would also be helpful. For example, if a user could specify that one sprite plays slightly ahead of another, she could easily create a chorusing effect. There could also be a "glide by" block for sound, paralleling the "glide by" block already implemented for visual sprites, and this would make it easy for users to create effects like scratching.

8.4.3 Make more compact interface with more integrated engine

The Squeak/Csound relationship is a powerful way to try various digital sound manipulations within the *Scratch* environment. It is an excellent way to prototype ideas in a testing environment. However, this environment is a bit problematic for a final product. The pipe is not as robust as would be needed when installing on multiple environments. And, at the least, Csound would need to be more hidden if running in the background. Also, the current Csound implementation allows for no error checking. Therefore, if an unexpected event occurs, Csound will disappear. Since it is impossible to predict what might happen in every situation, it is a bit worrisome to attempt to create a robust environment without error checking.

One solution may be to wrap the necessary audio processing functions in a C library and call them directly from Squeak. This has been the solution *Scratch* uses for its graphic effects.

8.4.4 Expand import/export facilities

Currently, the system supports importing .aiff files only. There is no export facility for sounds manipulated within the system. Already users have attempted to import .wav files. The import facility needs to import as many different types of files as we can support. Also, users need to be able to export their manipulated sounds outside of the environment easily and in multiple formats.

8.4.5 Provide more sounds

While the system is not intended as a way to manipulate synthetic waves, it can do this if synthetic waves are provided as samples. For users who want to create drum beats and guitar riffs, the system should provide appropriate audio samples for these as well. Perhaps there should be a folder of sounds with a subset of the MIDI specification.

In evaluation, users actually had interesting uses for the provided sine wave, which had been intended only for internal testing and not for users. Providing more waves that are short, have no beat of their own, and can easily be integrated with visual sprites could add power to the system.

8.4.6 Provide more demonstrative code

Right now there is only one true demonstrative script available for *SoundScratch*. It is an excellent script, written by Jay Silver, but it is complicated as an initial script to introduce users to the environment, and it is not enough as a general introduction for what is possible.

Users enjoy seeing scripts written on the fly with the "forever" and "mouse x" blocks that immediately show real-time interaction with the effects. It's not enough. The jump between these demonstrations and creating a script with a story or a unique and personal statement is too big and users have trouble making this leap. Demonstrative code would help address this.

8.4.7 Add support for higher level structures

Although the system should continue to support manipulating the sounds themselves, users will need ways to build high-level structures that make sense musically, are intuitive to them, and build on the framework of writing scripts in *Scratch*. With this in mind, we are looking at models of sequencers in other software and considering what parts of these interfaces are valuable for our own work and support the programming paradigm in *Scratch*.

References

Adan, V. (2005). Hierarchical Music Structure Analysis, Modeling and Resynthesis: A Dynamical Systems and Signal Processing Approach. Media Laboratory. Cambridge, MIT. **MS**.

Adobe Inc. (1990-2005). Photoshop CS. San Jose.

Aimi, R. (2002). New Expressive Percussion Instruments. Media Laboratory. Cambridge, MIT. **Master's of Science**.

Assayah, G. and Dubnov, S. (2004). Using factor oracles for machine improvisation. Soft Computing. **8:** 1-7.

Bamberger, J. (1979). Logo Music Projects: Experiments in Musical Perception and Design. A. I. Laboratory. Cambridge, MIT.

Bamberger, J. (1979). "Music and Cognitive Research: Where Do Our Questions Come From; Where Do Our Answers Go?" Division for Study and Research in Education: MIT **WP**(2).

Benbasat, A. Y. and Paradiso, J. A. (2005). A Compact Modular Wireless Sensor Platform. Proceedings of the 2005 Symposium on Information Processing in Sensor Networks (IPSN), Los Angeles, CA**:** 410-415.

Berry, R., Makino, M., et al. (2003). The Augmented Composer Project: The Music Table. Proceedings of the Second IEEE & ACM International Symposium on Mix and Augmented Reality (ISMAR)**:** 338.

Brosterman, N. (1997). Inventing Kindergarten. New York, Harry N. Adrams.

Cakewalk. (1987-2005). "Cakewalk." Twelve Tone Systems, http://www.cakewalk.com.

Chadabe, J. (1997). Electric Sound: The Past and Promise of Electronic Music. Upper Saddle River, Prentice Hall.

Costanza, E., Shelley, S. B., et al. (2003). Audio d-Touch: A Tangible User Interface for Music Composition & Performance. Conference on Digital Audio Effects (DAF), London.

D. Zicarelli, G., Yaylor, J., et al. (1990-2004). Max 4.3 Reference Manual, Cycling '74/Ircam.

diSessa, A. (2000). Changing Minds: Computers, Learning, and Literacy. Cambridge, MIT Press.

Farbood, M. and Jennings, K. (2004). Hyperscore: A Graphical Sketchpad for Novice Composers. IEEE Computer Graphics and Applications**:** 50-54.

Finale. (2003-2005). "Finale: Music Notation Software." makemusic! , http://www.finalemusic.com/.

FruityLoops. (2003-2005). "Fruity Loops: A full-featured sequencer." Image-Line Software, http://flstudio.com.

Gorbet, M., Orth, M., et al. (1998). Triangles: Tangible Interface for Manipulation and Exploration of Digital Information Topography. Proceedings of CHI, Los Angeles**:** 49-56.

Hiller, L. and Isaacson, L. (1959). Experimental Music. New York, Mcgraw-Hill.

Holmes, T. (2002). Electronic and Experimental Music. New York, Routledge.

Holt, J. (1967). How Children Learn. New York, Delacorte Press.

Ishii, H. and Ullmer, B. (1997). Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. Proceedings of CHI, Atlanta**:** 234-241.

Lefford, N., Scheirer, E., et al. (1999). "An Interview with Barry Vercoe." http://www.media.mit.edu/events/EMS/bv-interview.html.

Maloney, J., Burd, L., et al. (2004). Scratch: A Sneak Preview. Second Annual Conference on Creating, Connecting, and Collaborating through Computing, Kyoto**:** 104-109.

Manning, P. (2004). Electronic and Computer Music. New York, Oxford University Press.

McCartney, J. (1996). SuperCollider: a new real time synthesis language. Proceedings from the International Computer Music Conference (ICMC), Hong Kong.

McPherson, A. (2005). Interconnectable Blocks for Music and Audio Processing. Electrical Engineering and Computer Science. Cambridge, MIT. **ME**.

Merrill, D. (2004). FlexiGesture: A sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control. Media Laboratory. Cambridge, MIT. **MS**.

Meso (1999-2005). VVVV: A Multipurpose Toolkit. Frankfort.

Meyer, L. B. (1956). Emotion and Meaning in Music. Chicago, The University of Chicago Press.

Miles, B. (1997). Paul McCartney: Many Years from Now. New York, Owl Books.

Newton-Dunn, H., Najano, H., et al. (2003). " Block Jam: A Tangible Interface for Interactive Music." Journal of New Music Research **32**(4): 383-393.

Papert, S. (1980). Mindstorms. New York, BasicBooks.

Papert, S. (1993). The Children's Machine. New Tork, BasicBooks.

Paradiso, J. (2004). "Modular Synthesizer." http://web.media.mit.edu/~joep/synth.html.

Paradiso, J. A., Pardue, L. S., et al. (2003). " Electromagnetic Tagging for Electronic Music Interfaces." Journal of New Music Research **32**(4): 395-409.

Patten, J., Recht, B., et al. (2002). Audiopad: A Tag-based Interface for Musical Performance. New Interfaces for Musical Expression (NIME).

Propellerhead (2001-2005). Reason. Stockholm.

Puckette, M. (1997). Pure Data: another integrated computer music environment. Second Intercollege Computer Music Concerts, Tachikawa, Japan: 37-41.

Resnick, M. (1996). StarLogo: an environment for decentralized modeling and decentralized thinking. Conference on Human Factors in Computing Systems, Vancouver, ACM Press: 11-12.

Resnick, M., Bruckman, A., et al. (1996). "Pianos Not Stereos: Creating Computational Construction Kits." Interactions **3**(6).

Resnick, M., Kafai, Y., et al. (2003). A Networked, Media-Rich Programming Environment to Engance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities, MIT, UCLA.

Resnick, M., Rusk, N., et al. (1998). The Computer Clubhouse: Technological Fluency in the Inner City. High Technology and Low-Income Communities. D. Schon, Sanyal, B., Mitchell, W. Cambridge, MIT Press.

Risvig, F. B. (2005). Senior Technologist, LEGO. Cambridge: conversation at an MIT Media Lab open house.

Schultz, M. (2005). "1955 RCA Electronic Music Synthesizer." http://uv201.com/Misc_Pages/rca_synthesizer.htm.

Shannon, C. and Weaver, W. (1949). A Mathematical Theory of Communication. Urbana, University of Illinois Press.

Sibelius. (1998-2005). "Sibelius: Music Notation Software." The Sibelius Group, http://www.sibelius.com/.

Singer, E. (2003). Sonic Banana: A Novel Bend-Sensor-Based MIDI Controller. Proceedings on New Interfaces for Musical Expression (NIME), Montreal, Canada: 220-221.

Sony Media Software (2005). ACID Music Studio. Madison.

The National Association for Music Education (1994). The School Music Program: A New Vision. The K-12 National Standards, Pre-K standards, and what they mean to music educators. Music Educators National Conference, Reston, VA.

Tisue, S. and Wilensky, U. (2004). NetLogo: A Simple Environment for Modeling Complexity. International Conference on Complex Systems, Boston.

Trevino-Rodriguez, J. and Morales-Bueno, R. (2001). "Using Multiattribute Preduction suffix graphs to predict and Generate Music." Computer Music Journal **25**(3).

Vercoe, B. (1985). Csound: A Manual for the Audio Processing System and Supporting Programs. Cambridge, MIT Media Laboratory Music & Cognition.

Vercoe, B. (2005). control / audio signal dichotomy. J. Harrison. Cambridge**:** email.

Vercoe, B. and Ellis, D. (1990). Real-time Csound: Software Synthesis with Sensing and Control. Proceedings of the International Computer Music Conference (ICMC).

VirtualDJ. (2001-2005). "VirtualDJ: The Ultimate DJ Software." Atmoix Productions, http://www.virtualdj.com.

Weinberg, G., Orth, M., et al. (2000). The Embroidered Musical Ball: A Squeezable Instrument for Expressive Performance. Proceedings of Conference On Human Factors In Computing Systems (CHI), The Hague, ACM Press**:** 283-284.

Wishart, T. (1994). Audible Design. London, Orpheus.

Zicarelli, D., Yaylor, G., et al. (1997-2004). MSP 4.3 Reference Manual, Cycling '74/Ircam.

Appendix A: Python → Csound commands: *SoundBlocks*

```
Commands to Csound are 3 chunks long:

chunk 1: 0xc0 [cmd]     [param 1]
chunk 2: 0xb0 [param 2] [param 3]
chunk 3: 0xa0 [param 4] [param 5]

each parameter is 8 bits, holding a number between 0 and 255

COMMANDS:

1 : create instance
[param 1] = instrument #
[param 2]*128+ [param 3] = instance #

2: delete instance
[param 1]*128 + [param 2] = instance #

3: connect output to cable
[param 1]*128 + [param 2] = instance #
[param 3] = output # in instance
[param 4]*128 + [param 5] = cable #

4: disconnect output from cable
[param 1]*128 + [param 2] = instance #
[param 3] = output # in instance
[param 4]*128 + [param 5] = cable #

5: send sensor value
[param 1]*128 + [param 2] = instance #
[param 3] = sensor #
[param 4]*128 + [param 5] = sensor value

6: get value
[param 1]* 128 + [param 2] = cable #
```

Appendix B: Squeak → Csound commands: *SoundScratch*

```
- Create instances: i10 0 3600 [instance #]
- All commands: i[cmd type] 0 3600 [instance #]

- Set sound to:
[cmd (p1)] = 1001
[p4] = instance #
[p5] = ftable #
[p6] = length of sample

- Set sample speed
[cmd (p1)] = 1002
[p4] = instance #
[p5] = speed. Speed ranges from -8192 to 8191

- Set pitch
[cmd (p1)] = 1003
[p4] = instance #
[p5] = pitch. Pitch ranges from -8192 to 8191

- Set tempo
[cmd (p1)] = 1004
[p4] = instance #
[p5] = stretch. Reanges from -8192 to 8191

- Start sound
[cmd (p1)] = 1005
[p4] = instance #

- At end of sound
[cmd (p1)] = 1006
[p4] = instance #
[p5] = cmd:
0 - stop
1 - wait
2 - loop
3 - reverse

- Sound playing?
[cmd (p1) = 1007
[p4] = instance #
(Csound will return on Yoke NT 1: [224] [1] [x] [y])
x is 1 if sound is playing, 0 if sound is not playing
y is “don’t care”

- Stop sound
[cmd (p1)] = 1008
[p4] = instance #
```

```
- Rewind sound
[cmd (p1)] = 1009
[p4] = instance #

- Jump to
[cmd (p1)] = 1010
[p4] = instance #
[p5] = second to jump to. (- #s are counted back from the end)

- High pass
[cmd (p1)] = 1011
[p4] = instance #
[p5] = absolute value: range up to 1024. Filter from 50-5000Hz

- Low pass
[cmd (p1)] = 1012
[p4] = instance #
[p5] = absolute vale: range up to 1024. Filter from 5000-50Hz

- Reverb
[cmd (p1)] = 1013
[p4] = instance #
[p5] = absolute value: seconds of reverb /100. Up to 1000?

- Turn instance off
[cmd (p1)] = 1014
[p4] = instance #

- Reset all instances
[cmd (p1)] = 1015

- Reset instance
[cmd (p1)] = 1016
[p4] = instance #

- Initialize Csound
[cmd (p1)] = 1017

- Set live microphone
[cmd (p1)] = 1018
[p4] = instance #

- Delay
[cmd (p1)] = 1019
[p4] = instance #
[p5] = absolute value: mS for delay.
```