LogoRhythms: A Sound Synthesis and Computer Audition API for the Open Source UCB Logo Interpreter

by

Aaron Hechmer
B.A. Hampshire College, 1994

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

SUPERVISORY COMMITTEE

LogoRhythms: A Sound Synthesis and Computer Audition API for the Open Source UCB Logo Interpreter

by

Aaron Hechmer
B.A., Hampshire College, 1994

Supervisory Committee

Dr. George Tzanetakis, (Department of Computer Science and Music)
Supervisor

Dr. Daniel German, (Department of Computer Science)
Department Member

Dr. William Wadge, (Department of Computer Science)
Outside Member

Supervisory Committee

Dr. George Tzanetakis, (Department of Computer Science and Music)
Supervisor

Dr. Daniel German, (Department of Computer Science)
Department Member

Dr. William Wadge, (Department of Computer Science)
Outside Member

ABSTRACT

This thesis describes the construction, form, purpose and motivation for LogoRhythms, a sound synthesis and computer audition API intended to be used as a tool in the teaching of computer programming, computer science and associated skills. LogoRhythms is built into Berkeley Logo (UCB Logo), a contemporary open source Logo interpretter. In addition to serving as a user manual complete with program description and code examples, this work documents an exercise in experimental archaeology that traces the unfortunate shift in educational computing and personal computing in general from an emphasis of 'computer literacy' to one of 'user-friendly.' Arguments in defense of command-line and text based computing parallel those for computing as a tool for creative expression and are made in three ways: historical analysis, a new user-study and philosophical investigation. Programming is a widely learnable skill and debugging a useful skill transcending a utility limited to computer programs. Digital musical composition provides a perpetually renewable opportunity for custom software, underscoring that programming is a creative endeavor.

Table of Contents

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CHI - Computer-Human Interaction/Interface

FFT - Fast Fourier Transform

GUI - Graphical User Interface

HCI - Human-Computer Interaction/Interface

JPF - Just-Plain-Folk, ie. not institutionally trained

MIDI - Musical Instrument Digital Interface

UCB Logo - University of California at Berkeley Logo

WYSIWYG - What You See Is What You Get

Acknowledgement

Thanks to Adam Tindale for the MAX/MSP patch in Figure 1.1.

Chapter 1

LogoRhythms Introduction

1.1 Opening the Box

LogoRhythms is an audio synthesis, computer audition API built into U.C Berkeley Logo, an open source Logo interpreter. LogoRhythms adds sonic functionality to a language with a rich tradition in the teaching of computer programming and problem solving, even with primary school students. Writing programs that draw pictures via turtle graphics has always been a Logo mainstay application area [2]. With LogoRhythms, a program may produce both visual and musical output, underscoring a relationship of math and art and that computer programming is a creative opportunity.

Early versions of Logo were short on audio functionality, in part because hardware was lacking. Sometimes a function such as "tone" or "play" was included that enabled the machine to beep at a frequency specified as a parameter. Contemporary Logo implementations such as LCSI's Microworlds have far more advanced multimedia libraries [3]. This functionality, however, often focuses on high level manipulations such as using MIDI (Musical Instrument Digital Interface) to control a built-in synthesizer. Logo allows one to operate the computer at a fairly low level, at least low level compared to typical desktop GUI (Graphical User Interface) type applications. Analogously, LogoRhythms concentrates on low level audio manipulation, ie. manipulating arrays of audio data to build wavetables which in turn can be layered into compositions using programming devices such as procedures and recursion.

One might categorize the software most frequently used by computer musicians is into three types: graphical applications such as sound editors that often build their interfaces around an oscilloscope window showing waveform or spectrum, visual programming languages where functions are represented as graphical objects with pipes connecting them and, finally, traditional high-level typed-text languages.

Sound manipulation applications with well developed graphical interfaces run the gamut. Examples include: Snd, an open source, freely available sound editor from Stanford's CCRMA based on the emacs interface, including extendibility via Scheme (a Lisp dialog); the widely used Audacity; and perhaps topping the spectrum, DigiDesign's ProTools, a feature rich sound editor used in professional studios for mixing and final editing. When using these software tools, one almost always starts with some sound data, either recorded or generated elsewhere. The interfaces usually allow and memory management designed for work with many minutes of audio sampled at 44.1kHz or higher. Perhaps their greatest application is in mixing and arranging songs, though they are certainly usable to create short, novel audio snippets that, for instance, could be used as a wavetable in a synthesizer actuated by a MIDI enabled device such as a piano like keyboard. Functions such as filters and frequency transforms, usually FFT (Fast Fourier Transform), are often available. While filter parameters are configurable, they are not languages in which one would write a new filter from scratch nor do they tend to lend themselves to scripting or batch processing. While excellent for their task of audio manipulation and a good aid for teaching the physical principals of sound, they are not flexible programming tools. Musicians looking for programming tools in which to "code" sound synthesis are often drawn to a visual programming language, most likely either MAX/MSP or Pure Data (PD), an open source language idiomatically similar to MAX/MSP and maintained by one of MAX's creators, Miller Puckette [4] [5].

The programming environment, when first started, looks very much like the blank screen of a text editor, a clean slate waiting to be filled. However, the program instructions are laid out on the screen in an even less linear way than most structured text based

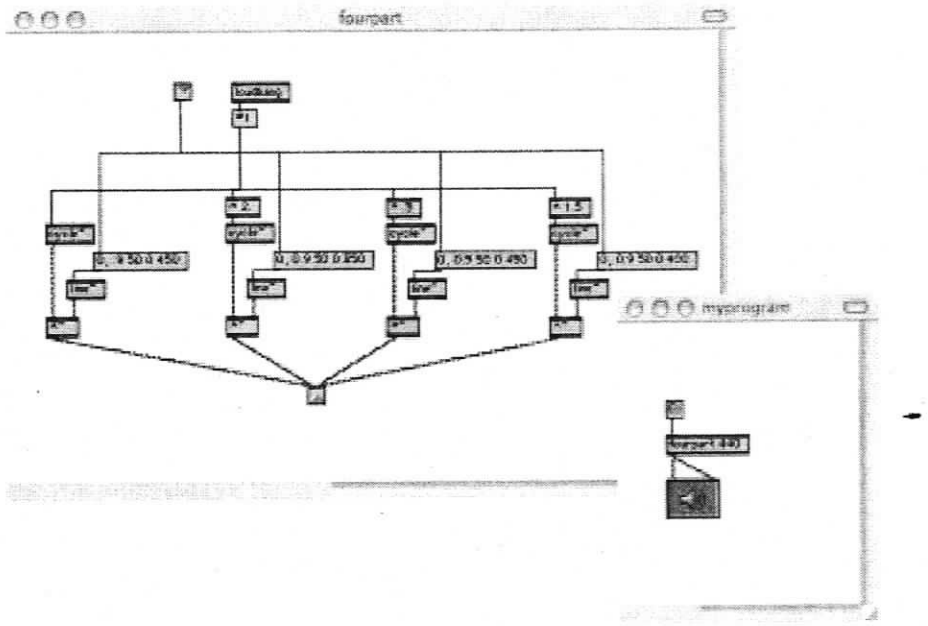

Figure 1.1. *A Four Part Harmony Program in MAX/MSP*

languages would be in a text editor window, Figure 1.1. Procedures, say for generating a sinusoid or the operation of addition, are drawn onto the screen inside of a box. The procedure names themselves can be chosen from a list of available primitives, somewhat freeing the programmer from needing to memorize the language's lexicon as well as facilitating exploration of available functions and their effects ("hmmm... I tried a *sin*, now what does this *cos* thing do?") Other boxes may contain numeric constants, have special functions such as toggle switches or a "bang" that sends a signal to trigger an event or display graphical information such as waveforms (an example of output) or envelopes (an example of input). Such boxed procedures are often functions in the sense of taking one or more arguments and returning some output. Graphically, these input parameters and output return values come and go to other boxes connected by a line, such as a box representing a digital-to-audio converter, ie. for play through the soundcard. For instance, addition takes two input lines and provides a single output line. The graphical metaphor

of programming as plumbing system, a schematic of faucets, pipes and sinks, has long proven itself via MAX/MSP and PD. Programs created this way are used in music heard on the radio, movie theaters, clubs, concert halls and public art installations and have been extended to applications such as controlling theater lighting; MAX programs have been written that take data read off external sensors such as anemometers used in a public art display in Seattle Public Library's Ballard Branch, processing the numbers as part of algorithmically driven musical composition. From the point of view of enabling musicians who may know no other programming language, PD and MAX/MSP are successful. Students without formal computer science training or knowledge in other programming languages regularly create nontrivial programs (known as patches in MAX/MSP and PD argot) that perform synthesis, time-frequency transformations, event handling and filtering. However this idiom is not the technology of choice for more general programming tasks: device drivers, web servers, 3D simulations of submarine telemetry are not written in this idiom. MAX/MSP cannot be written in MAX/MSP. Indeed MAX/MSP and PD offer hooks for extension via C/C++ for bolder explorations and customizations.

It is into this context that we introduce LogoRhythms, a music synthesis, computer audition API built on top of the functional flavored, typed-text paradigm of the UC Berkeley Logo interpreter. Logo and LogoRhythms are typed-text programming languages and as such creations are closer in relation to applications such as Lisp, C, Java or any other similar programming language than to applications such as word processors, spreadsheets or graphical musical editing software. Yet, Logo was created for general consumption and not just highly specialized gurus. Pragmatically, LogoRhythms has been created to teach programming through music. Theoretically, LogoRhythms has been created to provide a focal point in a debate comparing "user-friendly" against "computer literacy." Does user-friendly offer convenience at the expense of our own intelligence? While this is a clear question the answer is murky, awash in many shades of gray. For instance, is it worthwhile to learn underlying mechanisms if those details appear trivial to the task at hand? Are they not just an extra burden and distraction? In a marketplace where software companies

will be more than happy to do for you at a price, the question is worthwhile to ask despite its ambiguities. While LogoRhythms is offered as an educational tool for the neophyte to explore computer programming, my advocacy will strongly favor computer literacy over user-friendliness.

1.2 Chapter Overviews

This thesis document introduces the LogoRhythms API and its construction, presents LogoRhythms as a vehicle for teaching programming concepts to neophyte programmers and provides a historical and philosophical context into which to consider the debate between user-friendly and computer literacy that spawned languages like Logo and Smalltalk. The document concludes, as is customary, with suggestions for additional directions and research beyond that contained here.

The chapters breakdown as follows:

- *Chapter 2: A Tour of the LogoRhythms API* discusses the construction of LogoRhythms, the layout of the API including some short example programs and the limitations of UCB Logo and LogoRhthyms.
- *Chapter 3: Computer Science LogoRhythms Style* follows in the spirit of Brian Harvey's "Computer Science Logo Style" [6]. The LogoRhythms API is used in two non-trivial synthesizer applications. The examples demonstrate an introduction to binary search trees and hash functions in an audio application.
- *Chapter 4: Evolution and Obfuscation: A Case for Studying Antiques, Bicycles and Programming Languages* places LogoRhythms in historical context and argues why examining this language is a useful exercise in experimental archeology. Every year more and more libraries are available to the programmer. While undoubtedly useful for productivity, do these libraries add layers of indirection that hide the roots of the computer's operation, roots that are in themselves really quite simple?

- *Chapter 5: Flowers for Algorithm* examines ties between intuition and mathematics in a slightly different manner than usually offered by mathematicians and computer scientists, highlighting differences but focusing on the similarities. Writings from computer science and mathematics on intuition, proof, the aesthetic in mathematical discovery and constructionist learning are juxtaposed with phenomenological explanations of the embodied mind and sense borrowed from art criticism. This diversion hopes to provide thought on what is literacy and how design of artifacts is related to literacy, among other thoughts provoked.
- *Chapter 6: Conclusion (Open ended of course)* summarizes the implications of constraints and failure protection in HCI design against the goal of mechanistic revelation. As is customary, the conclusion makes some suggestions for further research.

1.3 A Brief Background on Motivations

This API came about from an immediate need. In the fall of 2004, I volunteered to teach an introductory programming class at the Honomu Computer Resource Center on the Big Island of Hawai'i. The center, in the town where I was currently living, was funded by the MacAurthur Foundation as one means to try to liven up the economy in an area hit hard by the departure of the sugar industry in the early 1990's. I wanted the class to focus on real programming in spirit and syntax, but I thought focusing on art and music related applications would make the after-school program more enticing while best matching my own interests in new media. The language I was to use had to work on a wide selection of equipment, ie. Apple OS9, MS Windows and Linux operating systems. The best choices, I thought, were UCB Logo or Squeak Smalltalk. Squeak comes with an excellent IDE, as has been the tradition in Smalltalk. However, I wanted something that focused on the text and the command-line and subsequently choose the more striped-down Logo interpreter. UCB Logo's drawback was lack of audio functionality. LogoRhythms provides that functionality.

Unfortunately, the class never materialized. But, it was not the center's directors' skepticism that proved the road block, although I was told, "I think you're trying to do too much, we had someone do a program last year and everyone was getting comfortable with the mouse by the end." When Logo was introduced, it's proponents at MIT successfully took it into schools where kids had had little or no previous exposure to computers (being the 1970's few kids did) [2] [7]. My failure was more pragmatic, the center lost its funding and closed its doors just prior to the class. Similarly, I tried pitching the idea among the local public primary schools. The principals I spoke with were enthusiastic. But when it came to actually assigning resources– a classroom with computers and an alloted time– the simple task was overwhelming. Overwhelming because these schools are chaffing under the US Government's "No Child Left Behind" requirements that leaves little time but to teach to the tests now required by the state and federal governments. [1]

I view the lack of field testing as my biggest disappointment for this thesis, creating a noticeable gap in the following pages. The schools, however, are bureaucratic institutes, more than somewhat cold to outsiders. 'Field testing' is best carried out by teachers or education students already in the fold of the institutions. For my part, I have kept the new code as close as possible to UCB Logo. LogoRhythms is open source and that it strives

[1] Euphemistically referred to as "No School Left Standing" by many educators, the federal program is viewed as a back-handed effort to discredit public education (and hence divert support to private institutions) and subsequently privatize what's left in its wake. By September 2005, for instance, over fifty two of two hundred and eighty public schools in Hawai'i alone failed to meet the testing criteria set by the government [8]. Upon failure, the local school board looses control and the state steps in. And what does the state do? Outside firms are contracted to restructure the offending school with bids in the range of $US250,000 per school. In contrast to Logo or Smalltalk, this restructuring involves quite a different and insidious approach to the use of software in the school system. Companies such ETS Pullium, a subsidiary of Educational Testing Services and EduSoft, a subsidiary of Houghton Mifflin offer enterprise level computer systems to the school. Databases contain the tests, the drills and the records of student's performance. In a PeopleSoft-esque twist, even letters to parents can be automatically generated and dispatched. Meanwhile, students sit at web interfaces running through the drills with apparently scant regard for conflicts of interest such as ETS writing both the standardized tests and training software.

to be available cross-platform and to as wide an audience as possible, although development has been carried out on a Linux based platform. Sometimes this comes at the price of performance and functionality, but not in the core functionality of providing a stomping ground to explore computer programming and computer science concepts. Therefore, my efforts are, for the moment, limited to an engineering effort and the philosophical justification behind the design decisions. LogoRhythms is provided free to the community with the hope that someone with stronger ties to the educational institution will find it useful.

Chapter 2

A Tour of the LogoRhythms API

The LogoRhythms API is mostly consists of primitives written directly into the UCB Logo interpreter using the C language. Some procedures are offered as part of UCB Logo's logolib, a library of procedures that appear to be primitives but are written in the Logo language. Functionally, the procedures can be divided into those that produce sound by simply reading through an array of floats until the final index is reached and wavetables, ie. short arrays that are read repeatedly as loops. It is not only possible but encouraged to create sounds using the array manipulation procedures and then use those arrays, or snippets of them, as wavetables.

This is an API with a fair amount of redundancy. For instance, the `sound` and `soundwt` procedures are superfluous to the `harmony` and `harmonywt` procedures. However, in keeping with the early design philosophy of Logo, the API is built to allow one to move from simple operations like `tone` that uses a sinusoid wavetable and a predefined amplitude envelope to `harmonywt` that allows the programmer to specify an arbitrary wavetable, multiple frequencies and envelopes. The format for arguments is conserved across all wavetable functions and is either frequency, amplitude and duration or frequency and then a list pair of amplitude and duration. Not only should consistency in argument format help avoid confusion but allows for a single procedure to return a list of arguments usable in a variety of different wavetable procedures.

In this chapter, I will briefly discuss the construction of LogoRhythms before elaborating on the API itself. Several short examples are given as suggestions for its use as well as

to highlight syntactical possibilities of programming in a functional style.

2.1 Construction

LogoRhythms relies heavily on two outside, open source pieces of software. The first is the UCB Logo interpreter maintained by Brian Harvey and colleagues at UC Berkeley [9]. UCB Logo provides the framework and the parsing functionality within which LogoRhythms is implemented. Of course, UCB Logo also provides an entire computer language with which to write many types of programs. The second piece of indispensable software is the PortAudio Portable Real-Time Audio Library maintained by Ross Bencina, Phil Burk and others [10]. PortAudio offers a C language audio API usable across most of the major platforms including Apple OSs, MS Windows and Linux/Unix. This API frees the application writer from needing to worry about sound card or platform specific nuances at the application level. PortAudio provides only the interface to the sound card and does not provide the synthesis or audition functionality that is organic to LogoRhythms.

LogoRhythms was initially implemented using version 5.3 of UCB Logo. Although releases of UCB Logo are infrequent, version 5.5 has recently been released. This document refers to LogoRhythms implemented in UCB Logo version 5.5. To facilitate merging, an effort was made to limit modifications of UCB Logo code. In addition to the UCB Logo makefile, modifications occur mostly in one of two places: globals.h and init.c. Both of these modifications involve making the names of the new primitives available to the interpreter. In globals.h the prototypes of the LogoRhythms procedures are given while init.c contains a multidimensional array named `prims` that lists the names of the logo primitives, the number of arguments each primitive accepts and the name of the corresponding C procedure to invoke within the interpreter. Unix specific thread code has been added to eval.c and is conditionally compiled. PortAudio files remain separate from interpreter code. The remainder of LogoRhythms is contained in separate files as follows:

- sound.h: Here one will find constant declarations, prototypes and two abstract data

types, EnvelopeData and OscillatorData. These new types are intended to be used as linked lists, ie. one EnvelopeData instance may point to another EnvelopeData instance, a necessary construction when building harmonies.

- wave.c: Source code for functions operating on arrays of data are mostly given here. In the LogoRhythms's argot, wave refers to a Logo array of floats; although, to the Logo programmer there is a simple number type that does not distinguish between floats and integers.
- sound.c: Procedures relying on wavetables are all found in sound.c. This includes a number of procedures that take arbitrary arrays of floats. The array argument is then played as a wavetable, ie. looped repeatedly.
- fft.c: Fast Fourier transforms between the time and frequency domains are kept here. This code is really just slightly modified versions of those found in Numerical Recipes [11]. The LogoRhythms's implementations use Logo data structures but the logic otherwise remains the same.
- audiofilter.c: Strictly speaking, this is not an official part of the LogoRhythms API. This file contains code for a biquad filter based on the implementation in Perry Cook's STK. The corresponding Logo primitive, `FILTERWAVE`, is not further discussed in this document and is not presently considered a formal part of the API.

There are two significant downsides from limiting modifications to native UCB Logo code. First, the limitation stops one from making improvements they wish existed. For instance, turtle graphics is kept simple in order to maintain cross-platform portability. Window redrawing behavior ends up being a bit quirky on Linux as well as lacking a method to add text to graphical output– a serious flaw when creating graphs of sound waves where one would like to keep track of amplitude and sample number. Making these improvements are important to LogoRhythms but presently beyond its scope. Second, all Logo data types are under the covers, kept in a massive structure called a `NODE` defined in the struct `logo_node`, a type regularly used in both linked lists and graphs. The lists and

arrays, even arrays of floating point numbers defining audio data, are in fact lists of NODE. In places this affects a significant performance penalty. To reiterate, LogoRhythms purports to be useful in teaching programming through sound. If one is seeking an optimally performing real-time audio API, this is likely not it.

The UCB Logo NODE does provide several advantages. First, it provides type checking functionality and new primitives which use it, which is all new primitives, immediately have access to the parser's error checking. Secondly, Logo is a descendant of Lisp and other than a number type, the list is the predominant data structure. Similar to Lisp, lists of commands can be executed or even used as anonymous functions. Even Logo's arrays really are little different in either operation or implementation from Logo's lists: both use NODE. The Logo interpreter has capitalized on the relation with Lisp by providing C macros such as `cons` , `car` and `cdr` with which to operate on lists of NODE within the C code. Unfortunately, this conservation of idiom only goes so far and semantics such as `caadddr` is not possible as it is in a Lisp interpreter such as Scheme.

2.2 API

The following is taken from the user manual for LogoRhythms found in a file called "SOUNDAPI" in the directory where UCB Logo has been unrolled and built. Logo is case insensitive and mixed cases will be found for the same command in the code examples below, ie. `make`, `mAke` and `MAKE` all reference the same primitive. The ubiquitous `mAke` is the assignment function; for instance, `MakE "a siNewAVe 220` assigns an array of floats representing one sinusoidal cycle to the variable `a`.

2.2.1 Activating

SOUNDON

Open a stream to the audio device. In other words, this needs to be called before using any sound producing method described here (except TONE). SOUNDON is sort of an audio

equivalent to PENDOWN in turtle graphics. The sound is on by default.

SOUNDOFF

The opposite of SOUNDON.

2.2.2 Procedures that Use a Wavetable

TONE *freq amplitude msec*

Make a sine wave tone with a fixed envelope. This function will call SOUNDON if necessary. TONE will output to the audio card, ie. it should make a noise:

```
ex. TONE 220 .9 1000
```

SOUND *freq [list of lists describing envelope]*

The audio is still a simple sine wave. However, the envelope can be specified as a list of value pairs. The first value gives the amplitude (0 to 1) and the second value the number of milliseconds to reach that amplitude (changing linearly).

```
ex. SOUND 220 [[.9 10] [0 800]]
```

This will produce a sine wave with frequency of 220Hz that reaches an amplitude of .9 (90% of full volume) in 10 msec before decaying linearly to silence over 800 msec.

HARMONY *[list of [freq [list of lists describing envelope]]]*

HARMONY extends SOUND. Sine wave oscillators can be combined, each using a frequency and envelope description similar to that of SOUND.

```
ex. MAKE "a LIST 220 [[.9 10] [0 990]]
    MAKE "b LIST 440 [[.4 10] [0 660]]
    HARMONY LIST a b
```

REST *msec*

Pause for a duration given in milliseconds.

TONEWT *wave_array [frequency amplitude msec]*

This function is very similar to TONE. Here one may specify an arbitrary array to use as the wavetable.

```
ex. make "a sinewave 220
    tonewt a [220 .9 1000]
```

SOUNDWT *wave_array [freq [list of lists describing envelope]]*

Similar to SOUND, SOUNDWT provides a finer level of control in defining an amplitude envelope.

```
ex. make "a squarewave 220
    soundwt a [220 [[.9 1000]]]
```

HARMONYWT *wave_array [lists of [freq [lists describing envelope]]]*

HARMONYWT provides the finest level of control over wavetables. It operates similarly to its sinusoidal cousin HARMONY. Note that the same wavetable, given by wavearray, will be used for all frequencies.

```
ex. make "a trianglewave 220
    harmonywt a [ [220 [[.9 1000]]]
                  [440 [[.9 1000]]] ]
```

SETTIME *envelope base_time*

SETTIME is essentially a normalizing function implemented as a library procedure. SOUND and HARMONY functions all use a list of amplitude and duration pairs such as [[.9 50] [0 950]]. SETTIME will take such a list and normalize it using *base_time* as a denominator. Its intention is to allow many different envelopes to be easily forced into the same time signature.

```
ex. make "new_envelope settime [[.9 50] [0 950]] 500
```

Here, "new_envelope is given the new values of [[.9 25] [0 475]] so that the durations sum to 500.

2.2.3 Procedures that Use Logo Arrays to Hold Audio Clips

SINEWAVE *freq*

Generate one cycle of sine wave values at the specified frequency.

```
ex. MAKE "wave SINEWAVE 220
```

TRIANGLEWAVE *freq*

Generate one cycle of triangle wave values at the specified frequency.

```
ex. MAKE "wave TRIANGLEWAVE 220
```

SQUAREWAVE *freq*

Generate one cycle of square wave values at the specified frequency.

```
ex. MAKE "wave SQUAREWAVE 220
```

NOISE *msec*

Generate msec milliseconds of noise:)

```
ex. MAKE "wave NOISE 1000
```

PLAYWAVE *wave_array*

Play, ie. send the data to the speakers, the clip specified in the wave_array argument. Playback occurs at 44.1kHz, give or take the processor's ability to maintain that rate.

```
ex. PLAYWAVE wave
```

COPYWAVE *wave_array num_copies*

Make num_copies number of copies of wave_array. So, if wave_array has a COUNT of 800, COPYWAVE wave_array 2 will return a new array of COUNT 1600. The following example should produce (more or less) one second of a 220Hz sine wave.

```
ex. MAKE "a SINEWAVE 220
    MAKE "a COPYWAVE a 220
```

WAVEENVELOPE *wave_array [envelope similar to that of SOUND function]*

This can be used to add an amplitude envelope to a sound clip (ie. what I've been calling a wave array). The form of the argument is very similar to that for SOUND.

```
ex. MAKE "a SINEWAVE 220
    MAKE "a COPYWAVE a 220
    MAKE "a WAVEENVELOPE a [[.9 50] [0 950]]
```

COMBINEWAVES *[wave_1 wave_2 ... wave_n]*

This function will combine its wave arguments into a new wave array with a COUNT equal to that of the longest of the arguments. The shorter arguments will simply be repeated as necessary to fill the space. Amplitudes are normalized automatically.

```
ex. MAKE "a SINEWAVE 220
    MAKE "a COPYWAVE a 220
    MAKE "b SINEWAVE 440
    MAKE "b COPYWAVE b 440
    MAKE "wave COMBINEWAVES LIST a b
```

ADDWAVEAT *wave_1 wave_2 msec*

This will insert wave_2 into wave_1 at msec milliseconds after its start. If msec happens to fall after the end of wave_1, a silence will be played as necessary before wave_2 commences. As with COMBINEWAVES, amplitudes are normalized, which means that the volume should be the same after addition as before.

CUTWAVE *wave_array [start end]*

Cuts a segment out of the wave array argument starting at "start" samples until "end" samples. One can use Logo's COUNT function to query for the total length of an array.

```
ex. MAKE "a SINEWAVE 220
```

```
;; COUNT a shows its length to be 200 samples
MAKE "a CUTWAVE a [0 50]
MAKE "a COPYWAVE a 880
```

REVERSEARRAY *wave_array*

Returns a new array with the order of indices reversed. REVERSEARRAY is implemented as a Logo primitive for performance. Still, large arrays may be slow to reverse.

```
ex. MAKE "drum READAUDIO "drumhit
    MAKE "reversed\_array REVERSEARRAY "drumhit
```

REVERSEWAVE *wave_array*

Syntactically this is identical to REVERSEARRAY; however, it is implemented in Logo as a library procedure. Although considerably slower, library procedures are more readily available to the student as a code example.

FFT *wave_array*

Transforms via fast fourier transform wave_array from a time domain to a frequency array zero-padding as necessary.

```
ex. MAKE "w COPYWAVE SINEWAVE 220 220
    MAKE "wf FFT w
```

IFFT *frequency_array*

Transforms via fast fourier transform from the frequency domain to the time domain. Using FFT followed by IFFT yields almost the same wave originally fed to the FFT.

```
ex. MAKE "w IFFT wf
```

NORMALIZEWAVE *wave_array*

Implemented as a library procedure, NORMALIZEWAVE will set the highest amplitude in wave_array to the max volume, adjusting all other values accordingly. This is particularly useful if COMBINEWAVES has created a clipped signal. To further adjust the volume, try the SCALEWAVE or VOLUME controls.

```
ex. MAKE "wave NORMALIZEWAVE wave
```

EVENWT *wave_array*

EVENWT is a library procedure intended to be used on wave arrays that will serve as wave tables. Since a wave table array is potentially played many thousands of times a second, it's useful if the first value and last value are nearly similar. Large differences may introduce a clicking or buzz. EVENWT chops off the end of the input `wave_array` until the first and last values are similar (but not necessarily identical). This does run the risk of clipping the entire wave!

```
ex. MAKE "wt EVENWT wt
```

DOWNSAMPLE *wave_array factor*

This library procedure subsamples a wave array by the factor specified in the second argument. It will return a smaller array (unless the factor is 1). In the example below, `w2` contains half as many sample points as `w1`. Since LogoRhythms's playback rate is fixed at 44.1kHz, `w2` will have a pitch twice that of `w1`.

```
ex. make "w1 TRIANGLEWAVE 440
    make "w2 DOWNSAMPLE w1 2
```

SCALEWAVE *wave_array delta*

This library procedure will modify the amplitude of wave_array by the factor delta.

```
ex. make "w COPYWAVE (SQUAREWAVE 440) 440\\
    make "w_quiet SCALEWAVE w .5
```

VOLUME *wave_array delta*

This library procedure is simply an alias of `SCALEWAVE`.

RECORDWAVE *msec*

Will record msec of audio into an array via the audio card's audio to digital converter. Of course, some sort of analog sound capture or producing device needs to be plugged into the audio-in, such as a microphone, electric guitar or tone generator.

```
ex. MAKE "w RECORDWAVE 1000
```

DRAWWAVE *wave*

This function is implemented in logo as a library procedure. So one may view it's code by looking for the file 'drawwave' in the logolib directory.

DRAWWAVE renders a drawing of the wave into a turtle graphics window compressing the time axis, by its best guess, to fit more or less across the width of the window. All sample points are accounted for to avoid any potential aliases due to subsampling. Note that if a new wave is drawn into the same window, it will also take up the whole width of the window but not necessarily have the same time axis scale, Figure ??.

```
ex. MAKE "w SINEWAVE 60
    DRAWWAVE w
```

2.2.4 Miscellaneous Procedures

READAUDIO *filename*

Reads an audio file and returns an array of the data. The file format is really just a subset of the Sun/NEXT .au format: stereo or mono 16bit linear at a 44.1kHz sampling rate. This procedure will give a warning if the file type looks incorrect, but will read in the file anyway. If you have trouble reading .au files, you may want to check its format and convert it to the above specs using a program like sox.

```
ex. MAKE "w READAUDIO "cool_song.au
    PLAYWAVE w
```

WRITEAUDIO *filename wave*

Write an array to disk. While this will write any Logo array to disk, it's really meant to be used with audio data. Because of the quantization in converting to 16bit PCM data (from the native floating point format), there's no guarantee that the exact same numbers will be read back in using READAUDIO as existed in the original array; they should, however, be close enough for the ear. The format is Sun/NEXT .au. However, the .au suffix is optional.

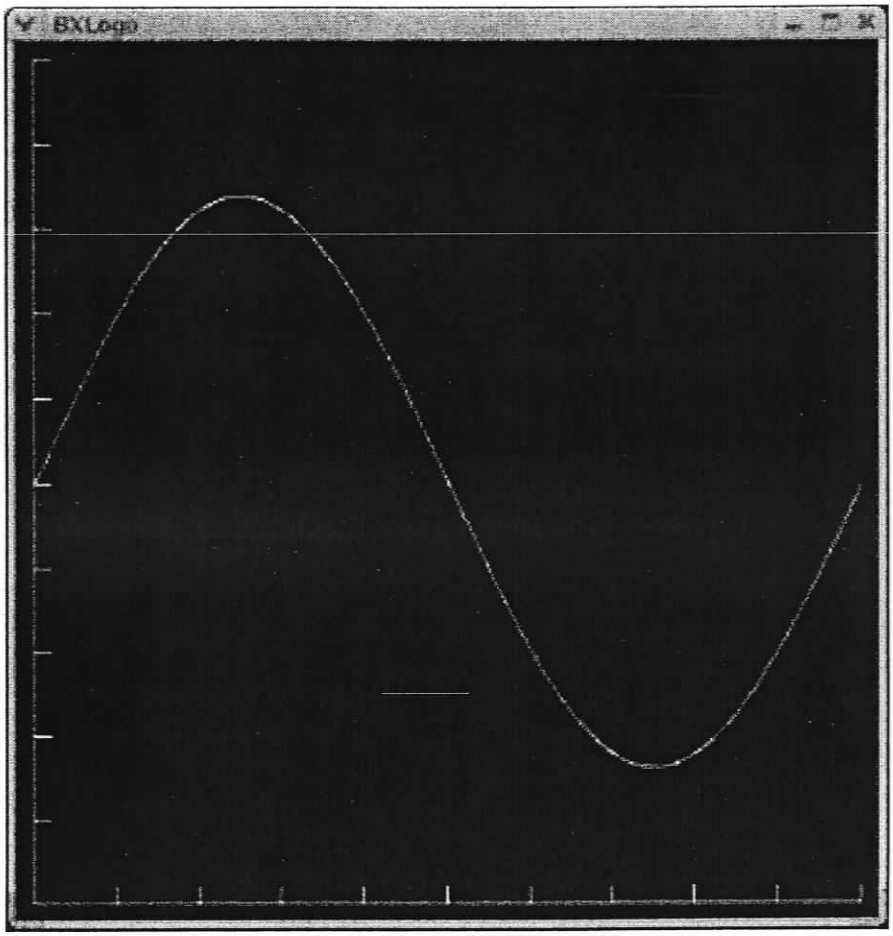

Figure 2.1. *Sinewave drawn in LogoRhythms*

```
ex. WRITEAUDIO "my_new_cool_song.au wave
```

BEATBOX *[instruction list] scale*

This library procedure acts as sort of a bucket for all of the sound producing procedures such as tone, tonewt, sound, soundwt, harmony, harmonywt and playwave. The procedures will simply be called in the order in which they appear in the instruction list supplied as the first argument.

```
ex. make "wv readaudio "drumhit
    make "beat [ [playwave wv] [rest 100]
                 [playwave wv]  [rest 100]
                 [tone 220 .6 500] [rest 100] ]
    beatbox beat 1
```

In this example, notice how every other instruction is a rest. The scale parameter will adjust these rests causing the beat to speed up or slow down. For instance, a scale of 2 will cause the beat to be played twice as fast; a scale of .5 will affect a beat half as fast.

2.3 Short Examples: Audio in a Functional Paradigm

LogoRhythms strives to be simple and the arguments' forms may be consistent, but there's no escaping the lengthiness and complexity of the required parameters. Lisp, Logo's progenitor, is often criticized for the morass of parenthesis marking the start and end of a list confronting the programmer. Neophytes are commonly warned to never endeavor in Lisp programming without an editor that will automatically match these parenthesis. Logo cleans up some of this mess, but lists, denoted by brackets in Logo, are still core to the interpreter's operation. Lists have many advantages. They are similar to arrays except that a single list may contain multiple data types and easily allows dynamic lengths, ie. elements may be added without explicitly allocating additional memory for the list. Lists may contain program instructions, are conducive to recursive programming, and, in Logo, may

even be accessed using an index (again similar to an array). Programmers already comfortable with abstract data types such as C structs or the classes of an object-oriented language may wish that some of the arguments were contained in such a data structure. Comparably long and complex parameter lists are found in Nyquist, a much more advanced audio language that uses Lisp syntax where Nyquist envelopes are also linear interpolations [12] [13]. The function is specified `(env` *t1 t2 t4 l1 l2 l3 dur*) where t1, t2 and t4 specify time segments and l1, l2 and l3 durations (dur is optional and defaults to 1.0). The t4 parameter describes the segments at the end of the envelope while the missing t3 is inferred thus freeing the user from keeping track of the total running length of the segment. The `env` procedure is a special form of `pwl` procedure that specifies piecewise linear functions and can be used for envelopes, glissando, filter specifications and more. The `pwl` function takes a variable and potentially long list of arguments: `(pwl` *t1 l1 t2 l2 ... ln*). In a functional language such as Logo, encapsulation and higher order functions can help address these messy arguments. Where C or C++ encapsulate data in a struct or class respectively, Logo encapsulates only at the function level, but in a way that is much tighter by trying to avoid global or class scoped variables. Several examples follow providing a demonstration of how the LogoRhythms API can be used paying close attention to cleaning up the arguments via encapsulation and higher-order functions.

Let's start with the following example creating a fourpart harmony of sinusoids.

```
harmony [ [440 [[.9 50] [0 450]]]
          [880 [[.3 50] [0 375]]]
          [220 [[.1 50] [0 450]]]
          [660 [[.05 50] [0 450]]] ]
```

The hard coded parameter is a mess! Thirty four brackets help separate lists nested four levels deep.

Encapsulation, in a broad sense, refers to the containing, even hiding, of information through scoping rules. Take for example the list that makes up the argument to HARMONY. This list can simply be encapsulated inside of another function. The list data is local to the second function and returned by it. Of course, it's not hard extend the functionality of

this second function such as adding parameters that modify the list to be returned. Here's a function called `fourpart` that will produce a list parameter for the HARMONY primitive.

```
to fourpart :freq
  local [a]
  make "a []
  make "a fput list freq      [[.9 50] [0 450]] a
  make "a fput list freq*2    [[.3 50] [0 375]] a
  make "a fput list freq*0.5  [[.1 50] [0 450]] a
  make "a fput list freq*3/2  [[.05 50] [0 450]] a
  output a
end
```

Now the HARMONY function can be called using the information encapsulated in `fourpart`, for example:

```
harmony fourpart 440
```

Ostensibly, this is a much clearer semantics. If the student programmer-musician also implements `fourpart`, even better. This first example demonstrates function composition of the form *f(g(x))* where *f(x) = HARMONY g(x)*, *g(x) = FOURPART x* and *x = 440*.

Templates are UCB Logo's device to allow the use of anonymous functions or, more specifically, lists of instructions. The real flexibility of templates begins to be realized when examining UCB Logo's APPLY function. APPLY itself takes a function as its first argument. The symbol "?" is called an explicit-slot and marks the parameters of the template function. The code:

```
APPLY [? * ?] [4]
```

will produce the product 16. Returning to the harmony example, consider the following procedure:

```
to sing :a.func :a.list
  ifelse (empty? :a.list) [  ]
  [ apply :a.func (list (first :a.list))
    sing :a.func (butfirst a.list) ]
end
```

This recursive procedure is very similar to map functions found in many functional languages. Its first parameter is a template. The second parameter is a list of arguments to

the anonymous function (ie. the template). It will recursively traverse over the list "a.list" applying each value in the list to the anonymous function "a.func." It differs from other map functions in that nothing, such as a new list, is returned since it's intended to be used with an IO affecting anonymous function. Our last code example uses `to sing` with the previous `harmony fourpart` demonstration to create a simple composition .

```
make "notes [440 494 554 587 659 739 830 880]
sing [harmony fourpart ?] notes
```

Do re mi fa so la ti do.

Using templates in this manner is similar to the use of lambda expressions in Lisp and the semantical distilling demonstrated here with LogoRhythms can be analogously accomplished using lambda expressions in a Lisp based language such as Nyquist

In this first example, the timing was left up to the definitions of the envelopes. This will always be true when using LogoRhythms's wavetable reliant procedures. However, templates of anonymous functions can simulate dynamic levels and time signatures of staff music.

First, several variables are created in the Logo workspace:

```
make ".base 1024

make "loud .9
make "normal .3
make "soft .1
make "silent 0
```

The `".base` variable provides the timing of a single measure in milliseconds. It also acts as a sort of global variable– Logo isn't a strict functional language that would normally completely avoid such scoping. Next, two wave arrays are created named buzz and breath.

```
make "buzz copywave squarewave 440 440
make "a copywave trianglewave 220 220

make "b noise 1000
make "b volume b .2
make "breath combinewaves list a b
```

The code snippet uses the `volume` procedure. Implemented as a logo library procedure, it runs a bit slower than the alternative syntax of `make "b waveenvelope b`

`[[.2 1000]]`, which is implemented as a primitive. However, `volume` provides semantic clarity and, in this example, the wave arrays `buzz` and `breath` will only be created once. The intention is to use the arrays with the `tonewt` procedure. One would normally think of a wavetable as a single period, or adequately long segment of a non-repeating waveform. The waves `buzz` and `breath` in fact correspond to approximately one second of output. Therefore, when setting the frequency parameter in `tonewt`, 1 will simply play the clip a single time; 2 will play the clip twice or twice the frequency. While this doesn't provide much control over frequency, for instance excluding partials, it simplifies the semantics in the proceeding example. A more refined version is suggested later.

The next step is to create some procedures taking care of note durations. The following two examples, `half` and `quarter`, provide the general framework which would extend to any duration one might want to name (sixteenth, triplet, etc...).

```
to half :func.tonewt :vol
  composenote func.tonewt vol .base/2
end

to quarter :func.tonewt :vol
  composenote func.tonewt vol .base/4
end
```

These procedures again make use of templates and anonymous functions, ie. instruction lists. The argument `:func.tonewt` has been named to suggest that the instruction list will contain the `tonewt` procedure. `Composenote` is a helper function that glues together the argument for `tonewt` and then executes the command via `apply`.

```
to composenote :func.tonewt :vol :t
   local [args]
   make "args list fput 1 list vol t []
   apply func.tonewt args
end
```

With this background work completed, the following syntax has been enabled:

```
whole [tonewt breath ?] normal
quarter [tonewt buzz ?] silent
half [tonewt buzz ?] soft
quarter [tonewt buzz ?] silent
sixteenth [tonewt breath ?] normal
quarter [tonewt buzz ?] silent
```

The subjective assessment of readability will be left to the reader. For example, were this example being used in a programming class for fourth graders, a not unreasonable expectation, the background functions such as `quarter` and `composenote` can be provided by the instructor if need be. This example could be made more flexible by using the `soundwt` procedure and allowing the frequency and amplitude envelope to be specified on a per call basis. To allow a wide range of frequencies, wavetable could be used in the more traditional sense– as a short (one or a few cycles) of the sound wave. The final syntax could take many forms, but with little modification to the example above, the following would be possible:

```
quartertone [soundwt buzz ?] [330 [[.9 50] [0 500]]]
```

A new function has been introduced, `quartertone`. Ostensibly this would be a modification of `quarter`. The absolute duration is given by the sum of the envelope segment lengths, `[[.9 50] [0 500]]`. The LogoRhythms logo library procedure `settime` is useful in converting this duration to a relative length; see the API note above.

2.4 Summary

To summarize, the LogoRhythms API can roughly be split into procedures operating on arrays of floating point numbers representing audio data and procedures that use wavetables. Of course, wavetables are simply arrays of floating point audio data, albeit usually just a short segment representing one or a few cycles. Since many of the procedures operating on these wave arrays, for instance `combinewaves`, are really working on linked lists of the *NODE* structure, they're relatively slow. Manipulation of arrays should occur ahead of playback time with the actual execution of sound taking advantage of previously composed wave arrays. A student composer can create timbres using the wave array manipulating procedures and then use those arrays as wavetables in the appropriate procedures.

From procedure to procedure, the design is graduated. Simple procedures such as `tone` allow easy entry into the API for the complete beginner, ie. low floor. These procedures

proceed to more complex and feature rich procedures such as `harmonywt` with some overlap between features. Given the right parameters, `Harmonywt` can produce the exact same note as `tone`.

Lists lend themselves to powerful and elegant programming solutions such as recursion, can be used as anonymous functions in the form of instruction lists as well as offering a growable container of mixed types. However, even given the effort to create clean, simple procedure names, list arguments can be unwieldy. Relying on functional language features such as templates and anonymous functions can greatly clean up syntax. Ideally, students may advance from being provided helper functions like `sing` and `quarter` to using anonymous functions and recursion in their own procedures.

In the next chapter, I will extend the examples above to demonstrate the introduction of common computer science problems using audio based programs. Two synthesizers will be built using binary search trees and hash look-up tables respectively.

Chapter 3

Computer Science LogoRhythms Style

3.1 Extending Harvey's Programming Primer

The previous chapter introduced the LogoRhythms API and provided some short examples of how one might use its procedures. A combination of carefully selected nomenclature along with instruction lists passed as anonymous functions was employed to syntactically approximate natural language. In this chapter, more moderately complicated applications of the LogoRhythms API is presented.

Brian Harvey's *Computer Science Logo Style Volume 3: Advanced Topics* shows Logo stretching its wings on topics such as artificial intelligence and compiler construction (a Pascal compiler) and non-trivial data structures such as trees [6]. First, I'll extend Harvey's example of balanced binary trees, including it as a central component in a digital synthesizer instrument. One might approach musical composition with LogoRhythms as a programming problem, ie. a program and its logic will dictate what notes will be played and when. Alternatively, performing the piece is simply a matter of running the program. The following example will allow a performer to use a LogoRhythms program to interactively perform music via the computer keyboard. In the second application, a different sort of instrument will be built, one that plays sampled music clips. For variety, a new data structure will be used, the hash look-up table. For computer scientists reviewing this document, these devices will be old familiar friends. For new programmers examining LogoRhythms, they may be an introduction to widely used data structures. The following

discussions assumes a basic understanding of programming, not necessarily an expert level.

3.1.1 Binary Trees with AlphabetSynth

The requirements for the first synthesizer are straightforward. First, we should allow a flexible mechanism for organizing a variety of sounds that may use either wavetables or sampled audio clips. Second, we need a mechanism by which the musician can select what notes to play and when. The first requirement will be met with a combination of balanced binary trees and anonymous functions. As for the instrument's interface, let's use the closest at hand: the computer keyboard. Of course, in keeping with the tradition of computer software and musical groups, the program will also need a name: AlphabetSynth.

In a nutshell, a balanced binary tree will provide a structure for storing synthesizer commands. The node's value, for instance, might be the base frequency of the intended note or perhaps the duration of a percussive hit. The tree also provides a rapid indexing scheme to speed searching out a note for a given key stroke.[1] Harvey introduces binary trees with a hardcoded example of telephone area codes and their respective cities. Here's an abridged version of his data: `[ [202 WASHINGTON] [404 ATLANTA] [808 HONOLULU] ]`. The data is given in a list of lists. While the tree will also be built using a list, this list isn't there yet. However, it should be noted that this first list has been numerically sorted by area code. Using a sorted list will allow us to construct a balanced tree with a short recursive function and without using rotation functions. The final form of this simple tree would be: `[ [404 ATLANTA] [[202 WASHINGTON]] [[808 HONOLULU]] ]`. Each level of the tree is a list of length three. The first element is the node, the second element a branch following the lesser value and the last element the branch following the greater element. Without re-presenting Harvey's code, his approach is simple. A pair of proce-

[1] This document's first audience, the academics evaluating its content, are, of course, intimately familiar with balanced binary trees. However, others may want to explore a simpler version of the data structure given in *Computer Science Logo Style Volume 3: Advanced Topics*, particularly Chapter 3 "Algorithms and Data Structures."

dures named `balance` and `balance1` take an input list and find its middle, ie. a median (or adjacent) value. The values to the left of this value will fall on the lesser branch while values to the right are greater. This process is then run recursively on each half of the subsequently shorter lists until all elements have been accounted for. In the code for AlphabetSynth, found in Appendix A, I've renamed `balance` and `balance1`, `btree` and `btreehelper` respectively.

Now, let's translate Harvey's area code data to something more in the spirit of AlphabetSynth. For instance `[808 HONOLULU]` might look like `[440 [soundwt :wave [[.9 50] [0 200]]]` where 440 is the frequency in hertz (an A in this case) that will serve as the node in the binary tree and HONOLULU becomes an instruction list that AlphabetSynth can potentially execute via Logo's `run` procedure. Similar to the list of telephone area codes, a list of `soundwt` instruction lists are generated on the semitone starting with a frequency that we'll provide as a frequency.

But let's put the binary tree list aside for the moment. `Soundwt` requires a wavetable parameter. Creating that wavetable is the heart of the synthesizer, as in sound synthesis, and will be performed via additive synthesis using the LogoRhythms primitives for array manipulation. FM synthesis is also a possibility and could be quickly implemented using a subsampling factor on a carrier wave that was modulated via a second wave. Since real-time FM is not implemented at the level of native C code, building an FM conditioned array for use in the wavetable procedures would likely provide the best performance. The additive synthesis used here simply sums waves, occurring in the procedure `synth`.

```
to synth

  local [ w1 w2 w3 freq ]

  ;; add a combintion of sinusiods
  make "freq 440
  make "amp 1
  make "w1 sinewave freq
  make "i 2
  repeat 2 [
    make "w2 sinewave freq * i
    make "i i*2
    make "amp amp/1.4
    make "w2 volume w2 amp
    make "w1 combinewaves list w1 w2
  ]
  make "w2 sinewave freq/2
  make "w2 volume w2 amp/2
  make "w1 combinewaves list w1 w2

  ;; add a combintion of trianglewaves
  make "freq 430
  make "amp .6
  make "w2 trianglewave freq
  make "w2 volume w2 amp
  make "w1 combinewaves list w1 w2
  make "i 2
  repeat 2 [
    make "w2 trianglewave freq * i
    make "i i*2
    make "amp amp/1.4
    make "w2 volume w2 amp
    make "w1 combinewaves list w1 w2
  ]
  make "w2 trianglewave freq/2
  make "w2 volume w2 amp/2
  make "w1 combinewaves list w1 w2

  make "n noise 1
  make "n volume n .05
  make "w1 combinewaves list w1 n

  make "w1 normalizewave w1
  make "w1 evenwt w1

  output w1
end
```

This procedure might be thought of as a sort of palette where waves are combined

at different amplitudes and frequencies until the musician-programmer is happy with the timbre. For those new to additive synthesis, perhaps the trickiest part of the procedure is the call to `normalize`– a slow function that first must find the largest value in the array and then adjust all other values proportionally, ie. requiring two passes. This is an important step, however, as the wave will otherwise be too large and clip when played, loosing the nuances so carefully added through these many additions. Now it's time to return to the list of notes with which to populate the binary tree, accomplished by the procedures `organ` and `organhelper`. The notes are semitones.

```
to organ :base :fade
  make "wave synth
  output organhelper (base * (1 / ln 2)) :fade :wave [] 0
end


to organhelper :base :fade :wave :data :i
   local [ env note freq ]

   make "env list [.9 25] lput :fade [0]
   make "freq (base * (power 2 i) * (ln 2))
   if less? 2048 freq [ output data ]

   make "env list freq env
   make "note lput env [soundwt :wave]
   make "note list freq note

   make "data fput note data
   make "i i + 1/12

   output organhelper base fade wave data i

end
```

These procedures require two arguments. The first is a base frequency, which in fact will be the lowest frequency. The second, a fade, describes the duration of the note's decay in milliseconds. The procedures' names are chosen to reflect the timbrel quality of the wave produced by `synth`. Therefore, a call to `organ` will return a list from the base frequency up to, in this rendering, 2048 in halftone steps. The list still isn't a tree; it needs to be sorted and then rearranged into a balanced binary tree. I've already discussed Harvey's balancing procedures, named `btree` and `btreehelper` in AlphabetSynth. Harvey, however,

started with a hardcoded sorted list of area codes. AlphabetSynth includes a sorting procedure, `sort`, that is based on the well known qsort algorithm and is not further discussed here. Additionally, a Lisp-like `cons` procedure is given for concatenating two lists. While it's trivial to implement, its ubiquity across most functional languages makes its absence as a primitive in UCB Logo surprising.

So to summarize, thus far we've covered the following functionality:

- creation of an array wavetable to use as the base timbre
- procedures that will create a list of LogoRhythms sound generating commands where each element is associated with a frequency on a 12 tone, semitone scale
- a procedure that sorts the list of commands by base frequency
- procedures– borrowed from Brian Harvey with some modification- that create a (fairly) balanced binary tree using a sorted list as input

Now, let's turn to the user-interface starting with the user and working back into the system and the binary tree data structure. The user is sitting at their computer. They will tap a key and get a tone, ie. there's a one to one mapping between pressing a key and a note. Some of the functionality will turn out to be generic for any possible synthesizer trees we create for the AlphabetSynth while some will be specific to this first tree, the list of semitone organ sounds. These two functionalities will be broken into two separate functions with generic functions being placed in a procedure named `startsynth`. Organ specific functions will go into a procedure named `organkeys`.

```
to organkeys :base
  make "base first :list.synth
  make "base base * (1 / ln 2)
  make "c ascii readchar
  output (base * (power 2 (:c - 97)/12) * (ln 2))
end
```

This procedure uses Logo's `readchar` procedure to handle the keyboard input. The character is converted to its ascii code and this code is offset such that the character 'a' becomes the number 1, 'b' is 2, 'c' is 3, etc.... These codes can then be used to calculate

a frequency. `Organkeys` then returns the frequency to the calling procedure to subsequently be used to lookup up the audio command in the binary tree. Here's a stripped down version of the procedure where the lookup will occur. Again, in Appendix A, one will find a more feature rich version that will be discussed later.

```
to startsynth :func.synth :list.synth :func.getkey :list.getkey
  local [ freq cmd b ]

  make "cmd []
  make "b btree sort (apply :func.synth :list.synth)

  forever [
   make "key apply :func.getkey :list.getkey
   make "cmd last lookup :key b
   run cmd
  ]

end
```

`Startsynth` is designed very much like the examples in the last chapter. The arguments to `startsynth` are templates, aka anonymous functions or instruction lists. Specifically, `:func.synth` and `:list.synth` will be the function and its arguments respectively that generate the list of synth timbres at different frequencies– in this example `organ` while `:func.getkey` and `:list.getkey` is the function and its arguments respectively that map keyboard keys to lookup "keys" used to find commands in the binary tree. `Startsynth` uses the former to generate the binary tree. Next, it enters into an infinite loop using Logo's `forever` procedure (all control structures are in fact procedures that take an instruction list as an argument). Keyboard strokes are read, mapped using `:func.getkey` template, queried for in the binary tree using `lookup` and finally the returned synth instruction list executed using Logo's `run` procedure.

And what does this all look like on the Logo interpreter command prompt?

```
? startsynth [organ ?1 ?2] [220 128] [organkeys ?] [220]
```

Remember that the '?' character marks the parameter slot in the template. In the case of `organ`, `?1` is the first parameter that maps to 220Hz and `?2` to the decay parameter mapping to 128ms.

Every instrument has its limitations and this one's no different. UCB Logo doesn't currently offer any threading mechanism. LogoRhythms introduces a basic thread procedure on Linux/Unix platforms. Threads, however, are not used in these examples. Each instruction list blocks the execution of subsequent instruction lists. Therefore, notes cannot be played concurrently. Additionally, the length of each `soundwt` is fixed. If the duration between two consecutively played notes is less than the duration of the note itself, the musician will experience a delay between their input and the feedback of sound– an upsetting of the one-to-one correspondence between action and reaction.

Up to this point, I haven't mentioned the lookup procedure for retrieving commands from the binary tree. Searching binary trees is similarly covered in many other sources including Harvey's example upon which AlphabetSynth builds. However, the code presented below for searching the tree does in fact deviate significantly from Harvey's. First, a number of simple query and predicate procedures have been created such as `isleaf?`, `getLessBranch` and `getMoreBranch` essentially as context appropriate aliases for basic list manipulation functions like `first`. But the bulk of the work is performed in `lookup` and `lookuphelper`.

```
to lookup :code :tree
  output lookuphelper :code :tree []
end


to lookuphelper :code :btree :closest
  local [ next less more ]

  make "this getNodeKey :btree

  if empty? :closest [ make "closest getNode :btree ]
  if equal? :code :this [ output getNode :btree ]
  if isleaf? :btree [ output closest ]

  ifelse less? :code :this [
    test empty? getlessbranch :btree
    iffalse [
      make "less getNodeKey getLessBranch :btree
      if updateclosest? :code :less first closest [
        make "closest getNode getLessBranch :btree
      ]
      test isleaf? getLessBranch :btree
```

```
      iffalse [
        make "closest lookuphelper :code getLessBranch :btree :closest
      ]
    ]
  ] [
    test empty? getmorebranch :btree
    iffalse [
      make "more getNodeKey getMoreBranch :btree
      if updateclosest? :code :more first closest [
        make "closest getNode getMoreBranch :btree
      ]
      test isleaf? getMoreBranch :btree
      iffalse [
        make "closest lookuphelper :code getMoreBranch :btree :closest
      ]
    ]
  ]

  output closest
end
```

I wish to make two comments about these procedures. First, while the strategy for searching the tree is normal– start at the root node and follow the branches left or right as necessary until a match is found– if no exact match is found `lookuphelper` will traverse the tree until it reaches a leaf. It will then return the *closest* match it has found in that traversal. In this way a given query is always guaranteed to return a command and, subsequently, a sound. Secondly, this is a subtly complicated procedure to debug and understand. The code contains ten different conditionals nested three levels deep. Furthermore, even short trees are hard to visualize as the lists they really are. Printing the list to screen is of marginal utility when a closing statement may contain five adjacent parenthesis. Here's how UCB Logo prints the binary tree for the first synth presented:

```
You don't say what to do with [[659.255113825739 [soundwt :wave
[659.255113825739 [... ...]]]] [[369.994422711634 [soundwt :wave
[... ...]]] [[277.182630976872 [... ... ...]] [[... ...] [...
... ...] [...]] [[... ...] [... ... ...] [...]]] [[493.883301256124
[... ... ...]] [[... ...] [... ... ...] [...]] [[... ...] [...
... ...] [...]]]] [[1174.65907166963 [soundwt :wave [... ...]]]
[[879.999999999999 [... ... ...]] [[... ...] [... ... ...] [...]]
[[... ...] [... ... ...] [...]]] [[1567.981743927 [... ... ...]]
[[... ...] [... ... ...] [...]] [[... ...] [... ... ...] [...]]]]]
```

No matter how much syntactic sugar we might mix into the LogoRhythms batter, this can be a hard function to debug. I took several hours to untwine the competing errors of several different syntax/logic errors, the output of each confounding my understanding and revealing of the others (of course, I used a much simpler tree as test data in the debugging process). This admission is made to temper expectations as I merrily proceed along advocating Logo and LogoRhythms to neophyte programmers. `Lookuphelper` demands concentration. A nice language may make it easier for a neophyte programmer to learn, but that doesn't always equate to it being trivial to learn. In my case, I had solid conceptual understanding of the algorithm's mechanism and was still left wrestling with implementation longer than I wished. Furthermore, the same mechanisms that allow very terse and clean code, particularly anonymous functions and recursion, can also make for some very dense and obfuscate constructions.

3.1.2 Hash Tables and FFTs with SampleSynth

Building on the idea of the AlphabetSynth, let's create another synth, SampleSynth. SampleSynth also uses the keyboard as the interface for playing a sound. Instead of using synthesizer tones generated by adding together wave forms, each key will be associated with a sample of recorded music. In general, LogoRhythms is designed to emphasize working with simple mathematical waves like sinewaves and squarewaves and the idea that, combined with envelopes, these simple building blocks can become any sound, at least in theory. Here, however, samples of recorded music are used to demonstrate what has been called, computer audition, ie. using the computer to hear the sound. More precisely, the computer is used to analyze the sound. In addition to changing the source of the sound from a synthesizer to samples, a different data structure is introduced for organizing the sounds. The binary tree is replaced with a hash look-up table, discussed further below.

First, we'll need some samples, at least twenty six or so to cover the alphabetic keys. We could use twenty six different songs, but this approach creates a stylistic problem, that a whole song is a very long and varied note, as well as a technical problem, that Lo-

goRhythms stores the data as uncompressed data. Each array of floats uses a considerable amount of memory. To create more abstract snippets and avoid any memory shortages, SampleSynth uses very short samples of a quarter second. Also, for the purpose of the example of computer audition, it will be helpful if the samples sound different, particularly in pitch. I've selected two pieces of music, *Permiteme* sung by Celia Cruz and J.S. Bach's *Toccata and Fugue in D Minor*.

The actual chopping and separating of the longer audio files permiteme.au and ToccataandFugueinDMinor.au is done by the audio format manipulation program sox, commonly found on many unix-like platforms including OSX and is not part of LogoRhythms. Calls to sox can be run via a shell script where 250ms are cut from the audio file along every five seconds of its length.

```
#!/bin/bash
for ((i=0; i<=100; i+=5));
do
  echo \${i};
  sox permiteme.au celia\${i}.au trim \${i} 0.25
  sox ToccataandFugueinDMinor.au bach\${i}.au trim \${i} 0.25
done
```

Each sample, or snippet, is written into a new .au file bearing the name celia__.au or bach__.au where __ is a number indicating where the snippet was clipped in the original file. With these snippets prepared, it's time to move back to Logo code. The full code for this example is included with LogoRhythms in the files samplesdb.lg. The next task is to get the filenames of the snippets into a list. It's not impossible to read a directory's contents in UCB Logo, but it's done using the `SHELL` procedure such as `make ''files shell ''ls` which uses a unix specific command `ls` to list the file names. To make life simpler, albeit less elegant or flexible, the snippet file names are hardcoded into the example file samplesdb.lg and stored in a list named `files`.

```
.
.
make "files fput "samples/celia5.au  files
```

```
make "files fput "samples/bach30.au files
make "files fput "samples/bach95.au files
make "files fput "samples/celia60.au files
.
.
```

In the Samples DB, each keystroke will play one sample. But which sample to play? And how to associate a given sample with a given key? We'll need to know something about the sound that is the snippet. What sort of attributes can we use when talking about these snippets? We could use length. But, the snippets have all been chopped to 250ms. We might say that some of the snippets sound like classical organ music and the other set like salsa. We might note that some of the snippets have a higher tone or pitch that others. It is in fact these two descriptions that interests us in building the Samples DB. What sort of instrument is being played and what is the predominate pitch of the sample? Ideally both of these question with an answer that can be expressed numerically. In the AlphabetSynth we knew what the fundamental frequency of each note because we created the note around that frequency, it then being used to organize the notes within the binary tree. The instrumentation, or timbre, and the pitches of the Cruz and Bach samples are a complex mix of frequencies. SampleSynth will work in reverse from the approach used in AlphabetSynth and instead start with the sound in the sample, then identifying a dominant frequency with which to label it. To summarize:

1. identify all of the frequencies associated with a snippet
2. select a single frequency from this frequency fingerprint that stands out
3. use the dominant, stand-out frequency to index the snippet in a hash look-up table

Here is the code that first runs when the the Samples DB is started:

```
print [making calculations]
make "db sort measurefrequency getaudiodata files
make "db listtoarray db

print [enter a key a to z]
```

```
forever [
  make "key ascii readchar

  make "key hashfunc key count db
  playwave last item key db
]
```

The first three lines create the database of snippets. The rest of the program is an endless loop that takes input from the keyboard, looks up the associated sound snippet and then plays the sound. The most exciting action, as far as mathematically describing the sound, starts in the procedure `measurefrequency`.

```
to measurefrequency :audio
  local [ peak ]

  if empty? audio [ output [] ]

  make "s fft first audio
  make "peak peakdetector spectrum s

  output fput (list peak first audio) measurefrequency butfirst audio

end
```

This procedure takes the audio snippet as an argument and then applies LogoRhythms's `FFT` procedure, a fast fourier transform. The transform that occurs is from time domain to frequency domain. In the time domain, each sample is associated with an amplitude, ie. volume. How much, how loud, is the sound at a given point in time? In the frequency domain, our data can now tell us what frequencies are present and at what intensities at a given window of time, an admittedly abstract notion since frequency requires time. So in reality, it's frequencies over a sliding window in time. If one plays the wave array returned by an fft, it will probably sound like a lot of static. LogoRhythms also provides an `IFFT` procedure, inverse fast fourier transform, that will return the frequency domain data to time domain data, sounding as expected when played through the speakers. While the fft accomplishes the transformation, the result is a complex number. It would be easier to think of the data in terms of how much of each frequency is represented and to think of

those frequencies in terms of cycles per unit time such as hertz, the same unit used for arguments to procedures such as `SINEWAVE`. LogoRhythms provides a Logo library level procedure called `SPECTRUM` that makes the conversion to the magnitude of frequencies in the samples expressed in hertz.

Finally, with the spectrum of frequencies in hand as the output of the `SPECTRUM` procedure, a single dominant frequency is found using the SampleSynth's `peakdetector` procedure. The peak detection strategy simply scans the data looking for the tallest peak. One slight twist, however, is to not simply look for the tallest single sample, but bin consecutive samples, done in the procedure `getwindowvalue` and then measure the tallest bin. In this way, one can get a rough measurement not just of the height of a peak, but also its width.

With the spectral analysis complete and a single frequency identified to characterize each audio snippet, it's time to get to the business of organizing the snippets in a searchable data structure, in this case a hash table.

A hash table is a data structure, usually an array, where the location of a given piece of data, say a number or some text, is related to the content of data. For instance, imagine an array of length 10 that will hold the numbers 1 through 10. So that we will immediately know where the number 5 is located, it will always be placed in the fifth index of the array. Of course, this means that only one instance of the number five may be stored. A slightly more complicated hash table might be built to hold the letters A-Z where A will always be in the first index, B in the second, C in the third and so on. This is also the case in SampleSynth. The fft is performed with the data in a Logo list. The code `make "db listtoarray db` converts the list of snippets and the dominant frequencies used for their labels into an array where members can be found with a numeric index. To find the specific index, a hashing function is used. In the general case, the input of the hashing function may be a number, a reference or text. For instance a well known hashing function returns the sum of ascii values for the characters in a string modulus the number of indices in the hash table. The key calculated from this function will be unique and normalized to

the size of the array holding the key's values.

The hashing function used here is even simpler. First the array holding the snippets is sorted by the value of its peak frequency. When the user types a key, SampleSynth obtains the ascii value, a numeric value associated with a character, for instance an "a" has an ascii value of 97. This value is brought into range of the array indices by subtracting 96. The result is the array index holding the sample to be played. Here's that explanation expressed in Logo:

```
to hashfunc :key :length
  local [ idx ]
  make "idx 1

  if less? 96 key [ make "idx key - 96 ]

  ;; keep in bounds
  if less? length idx [ make "idx length - 1 ]
  if less? idx 1      [ make "idx 1 ]

  output idx
end
```

Finally, once the snippet has been returned from from the hash table it is played with the `playwave` procedure. Afterall, it is just a wave array of sound data.

To summarize, this chapter has attempted to introduce two important data structure and indexing-searching strategies widely used in computer science by presenting them in audio applications build using LogoRhythms. The algorithmic topics take their context in two different computer instruments, each using the computer keyboard to allow the musician to actuate the sound.

AlphaSynth generates a tone based on a desired fundamental frequency. In other words, one decides what pitch the tone will have and then embellishes color around that pitch with varying combinations of harmonics, partials and wave types like sinusoids or triangle waves. The Samples DB works the other direction. Audio snippets are harvested from pre-recorded music. Using LogoRhythms's `FFT` and `SPECTRUM` procedures a peak frequency is associated with the snippet to be used as a lookup key in a hash table.

While one might be tempted to proclaim, 'Voilá!,' LogoRhythms is not intended to be a

final solution to anything. For sure, there are far better synthesizers available if one's only goal is performance. LogoRhythms is a user-friendly computing language. Compared to a sound editor application, it can provide a low level entrance into application writing. Having a language is not enough! Dissatisfaction and a drive to tamper, a curiosity to dissect and rearrange are also necessary. For starters, AlphaSynth and SampleDB can be tuned, just like a musical instrument. One new to Logo, LogoRhythms or computer programming might start by changing the synthesizer in AlphaSynth, for instance with a different combination of waves added together in the `synth`. While a student-programmer may be interested in extending the instruments, the data structures and algorithms transcend the musical application presented here. The programs could be modified to fit other purposes such as the exciting task of indexing telephone numbers by area code, as suggested by Harvey. Reuse and modification is key in the work-day aspect of software engineering and problem solving with computer applications. In that sense, the ability to modify, dissect, break and extend provides a strong argument for open source frameworks in educational computing. Learning something that's hidden from view with intellectual property miserliness and the intent that others cannot learn it or copy it should be regarded as anathema.

Whether with Logo or any other handy language, go forth, break and create.

Chapter 4

Evolution and Obfuscation: A Case for Studying Antiques, Bicycles and Programming Languages

4.1 Introduction

Donald A. Norman, frequent contributer to the computer-human interaction literature and former Apple Fellow, states that "good design" can be boiled down to four principles: (1) visibility, (2) a good conceptual model, (3) good mappings and (4) feedback. The principles are enumerated, along with seven measures for achieving them, on page fifty two of *The Design of Everyday Things* [14].[1] And yet on page one hundred and eighty five, Norman describes an ideal computer of the future as "invisible."

> The point cannot be overstressed: make the computer system invisible[14].

The following essay will tackle this contradiction, suggesting a resolution by way of another design principal not on Norman's list: transparency. Here, transparency refers to the revealing of mechanical causality through design considerations. Transparent design

[1] The seven measures: "Tell what actions are possible", "Tell if system is in desired state", "Determine mapping from intention to physical movement", "Perform the action", "Determine mapping from system state to interpretation", "Tell what state the system is in" and "Determine the function of the device".

may also strive to allow better access to historical antecedents, remnants of design giving insight into an artifact's evolution.

4.2 Mechanical Transparency on Large and Tiny Machines

Norman argues that it was the task that needed to be visible, not the machinery. He creates two categories for interacting with the computer, "command mode" and "direct-manipulation mode." Command languages are an example of command mode. Direct-manipulation mode includes, for example, video games, spreadsheets and text editors. Having indulged in this categorizing, Norman immediately goes on to make some paradoxical claims about direct-manipulation mode.

> But direct manipulation, first person systems have their drawbacks. Although they are often easy to use, fun and entertaining, it is often difficult to do a really good job with them. They require the user to do the task directly, and the user may not be very good at it. Colored pencils and musical instruments are good examples of direct manipulation systems. But I, for one, am not a good artist or musician. When I want good art or music, I need professional assistance. So, too, with many direct manipulation computer systems.

Are direct manipulation devices, including applications, "fun and easy" or "difficult?" Perhaps both, rendering the categorization far less helpful in the search for an ideal. Norman suggests that such interfaces may require professional assistance. Yet, this is the mode of spreadsheets, word processors and video games. More typically it is command mode and esoteric languages that have fallen to the purview of professional syntactic stunt people. Musical instruments are given as an example of a direct-manipulation, domain of the professional device. The computer is a musical instrument and electronic music can be programmed using command languages, the essential command mode.

Justifying Norman's logic is less fruitful than the observation that there really isn't so much difference between these modes as Norman might want us to believe. If musical

instruments are direct-manipulation devices and computers are musical instruments that can be played via command mode, perhaps command mode should really be considered a direct manipulation mode? Norman's copout of this paradox? Make the computer invisible, reduce it to nothing more than a platform for heavily constrained task specific tools, remove programmability and flexibility, diet away the challenge of learning the tool but in the process loose the rich fat of adaptability to specific problems and alienate the end-user from the underlying mechanisms by which the machine operates.

Making the computer invisible begs the question of confusion since the units of computation, bits, bytes, logic gates, etc... are so small that they're already impossible to see. The outer perimeters of computers may become extremely, even conveniently, small. But it's also likely that their inside space continues to become larger and larger as more and more memory is squeezed onto chips, or at least not shrink from the spaciousness already attained. And hence a significant dilemma of computer-human interaction, whether command language based, wysiwyg point and click or some synergism of the two, is that most of what's in the computer will not fit on the screen during a single moment. In comparison, most musical instruments are visible... and audible. Musical instruments provide ample feedback including haptic. They generally provide a magical and direct relationship between the visible, audible and haptic. Electronic music, unsurprisingly, easily violates this relationship visually, the strings too small to be seen.

A bicycle is an excellent subject for questions.

So wrote Jean Piaget in *The Child's Conception of Physical Causality*[15]. The famed psychologist was contemplating how we people come to build our mental models explaining mechanistic causality, in this case how a bicycle works. And, the reason a bicycle is so excellent?

All the pieces of this mechanism are visible.

Piaget and Norman would seem to agree that visibility is an aid in the understanding of causal mechanisms.[2] For Piaget, the visibility of the mechanics made for experimental convenience. Piaget reasoned that all adults can name the key components of a bicycle (wheels, chain, handlebar, etc...) circa 1930 and describe the role they play in the bicycle's movement. In fact Piaget put the age at which the causal roles are obvious at around 8 years old, at least for the French boys with whom he worked[15].[3] However, his interest was in tracing the development of such intuitive obviousness. Visibility alone apparently is not enough early in a child's development. His four year old subjects could see and even name the parts of a bicycle. Yet, their causal explanations for movement fall short, eg. there are motors in the spokes or currents of air or water inside the tires propelling the bicycle forward. Still implicitly, the experiment supports the importance of visibility in understanding a design and more so demonstrates that some developmental change, which I'll just simply call "experience," actually leads to elimination of mysterious, hidden causality, such as a stiff breeze inside the tires. [4] The shape and movement of controls should map clearly to their functions, eg. turn the handlebar right and the bike turns right, mechanical causalisty reflected in the design. Norman would seem to agree with such logic, particularly in the design of doorknobs and sink faucets, favorite crusades of his, but less so with the computer.

Is a piano easy or difficult to play? It's almost immediately obvious how to get sound out of a piano via the keyboard, but it can require decades of practice before Carnegie Hall stands in ovation. Or, as a different measure, I cannot match via the piano the complexity with which I can hum, whistle or hear music in my head, beit a recording of another piano

[2]'Accessibility' is probably a suitable synonym for visibility here. The important essence of visibility being accessibility to the senses, ie. in apposition to invisible, hidden, unknown.

[3]Piaget suggested the age for girls was later as they had less interest in bicycles. I'm suspicious that such a claim would be born out with the young daughters of my cycling fanatic friends.

[4]Piaget suggests that at some early age no amount of patient parental explication, like cutting open a spoke, will suffice as the mind's just not ready. So perhaps the question is what role does experience play in coming to solve the mystery... the old nature vs. nurture silliness.

player or a simple day dream improvisation. The instrument demonstrates easy access (low floor) without prematurely stifling room to grow (high ceiling). With nominal training, thirty minutes perhaps, one can learn enough to reproduce the simple 2-5-4 or 1-4-5 chord progressions characteristic of much pop music- pianos are used successfully in a wide variety of artistic applications ranging from simple chords to concurrent playing of bass lines, chords and melodies by a single player. Beyond the question of playability and my own lackluster tickling of the ivories, I feel comfortable in describing how a piano produces a sound via levers, hammers and strings. A cursory glance inside an open grand piano reveals the basic mechanisms. With an open face plucked string instrument such as a guitar, I may even be able to visualize the nodes on the string corresponding to the harmonics of its base frequency. These instruments may not quickly avail their users to succeed in a certain musical task, but their mechanistic causality is not elusive in its main components, a qualifier added in recognition of less obvious nuances: dryness of the wood, interaction of two concurrent tones, and the infinitum of complex systems. When I first saw a gamelan as a teenager, I essentially had instantaneous understanding of its mechanistic causality, enough so to "play" notes even though I will not be joining a Balinese orchestra anytime soon.

Making a design analogy between the physical mechanism of a computer such as the electric charges on doped silicon and the circuit layout of logic gates to a bicycle or a musical instrument would be quite difficult if based on visibility. One could build the computer to a very large scale, say where each circuit is a few millimeters wide. Of course, such a computer might be so large that the big picture would extend off the horizon. Engineering efforts have mostly been striving for the opposite transformation in size like reducing one bit of information to something the size of a single proton– a bit encoded by the spin direction of the particle. A second approach might employ a sort of microscope. A microscope, similar to the fantastically large computer, would improve local detail but again at the expense of the bigger picture. Still this could be an improvement of sorts. The computer works on patterns and much of the size of a circuit's design results from redundancy,

eg. holding many bits of data. Revealing a single piano string provides a correct starting model as to how each string will operate. Revealing two strings will suggest a relation of string diameter to pitch, wider strings producing lower tones– a revelation reminiscent of induction.

While the boards, buses and chips of today's computers are still visible, computation already occurs at an invisible level. The sensibilities of intuition are deprived of their old crutch, eye-level seeing. Magic and the unexplainable waft from the miracle of miniaturization and its elusive mechanisms, like the motors in the bicycle spokes of Piaget's four year olds.

Most of us use our sight sense when negotiating the computer. The commonest user-interfaces, command mode and wysiwyg, depend on it. But the images have been transformed any number of unknown times between the chip and our retinas. Transformations similarly occur for auditory or haptic input-output. These transformations bridge magnitudes and coding schemes from binary electric charges to Latin alphabet glyphs. The tranformations' designs define the mappings between mechanistic causality and the senses. Mapping and visibility, such as used by Norman, may provide thought provoking categorizations. But they aren't strictly exclusive of each other. I can watch somebody drive a car and the mapping between steering wheel turns and car turns will aid my understanding in use of the device. Mapping is even more closely related to feeling, not necessarily haptic, ie. touch, but of the act-and-react, experimentation kind as in "feeling one's way through a problem." Reach out and affect some inertia. How did the state of the system change?

Let's say our interest is to understand the physical mechanisms of the computer. Not such a strange desire. The semantic content on a disk is as firmly encoded by magnetic charge as ink set on paper. Tasks, like producing and reproducing electronic music, are physical and might be explained as a relationship of physical mechanism. These mechanisms can be understood by appealing to intuition's acceptance of causality, enumerations of a sequence of acceptable relationships. A widely used user-interface standard for affecting this physical mechanism is assembly code, a catchall for language that widely varies

across hardware but in every case closely captures the changes that must occur at the level of the physical computer. Assembly code not only provides an interface for issuing changes to the physical machine on a one-to-one basis, but provides a nice outline of causal relationships. Assembly code, more than other programming languages, satisfies the goal of clear mappings through the transformations between the controls and the changes in the underlying device. Here is the assembly code for the program that adds one and one (1 + 1) on my Intel machine as translated from a short program written in the C language.

```
.file "test.c"
   .text
   .globl main
   .type main,@function
main:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   andl $-16, %esp
   movl $0, %eax
   subl %eax, %esp
   movl $2, -4(%ebp)
   movl $1, %eax
   leave
   ret
.Lfe1:
   .size main,.Lfe1-main
   .ident "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

The mapping *is* clear. An instruction such as `movl` literally moves electrical charges from the memory at `%esp` to `%ebp`; the movement is as real as if one had two piles of

oranges, moving the fruit between piles to count off arithmetic operations. The same program could be accomplished by the manual toggling of physical switches. The program *is* readable, leveraging our expertise with the symbols of a widely used natural language. Perhaps the mapping to physical mechanisms of the computer doesn't get any better than this. Mapping toward the natural language skills used outside of computing does. The C program from which this assembly code was generated is given below. The C code more closely follows familiar algebraic notation, ie. a standard of sorts introduced early to most primary school students.

```
int main() {
   int r;
   r = 1 + 1;
   return 1;
}
```

What has been gained in the abstraction, the transformation between the assembly code and C code? The program is terse. More can be accomplished with less code. The program's operation is clearer. What is lost? The cost is clarity of mechanistic causality.

Programming languages may be regarded as the domain of the highly trained, the professional, the guru. But they weren't invented to benefit computers. Computers existed before programming languages and can operate fine without these levels of indirection and abstraction. For instance, the Buchla is an early analog synthesizer where programs are created by patching together wave generators, amplifiers and filters using cables in the manner of an old fashioned telephone switchboard, Figure 4.1. Text based, terminal readable programming languages make computers easier to use in a way that, by extension, increases their utility. Ease and utility exist in a precarious balance. A slice toaster is easier to use than a hibachi, but less versatile. With the invention of programming languages, little utility is lost. Translucency of the mappings to mechanistic causality becomes cloudier, but everything the computer could previously do, it can still do. The increase in speed and

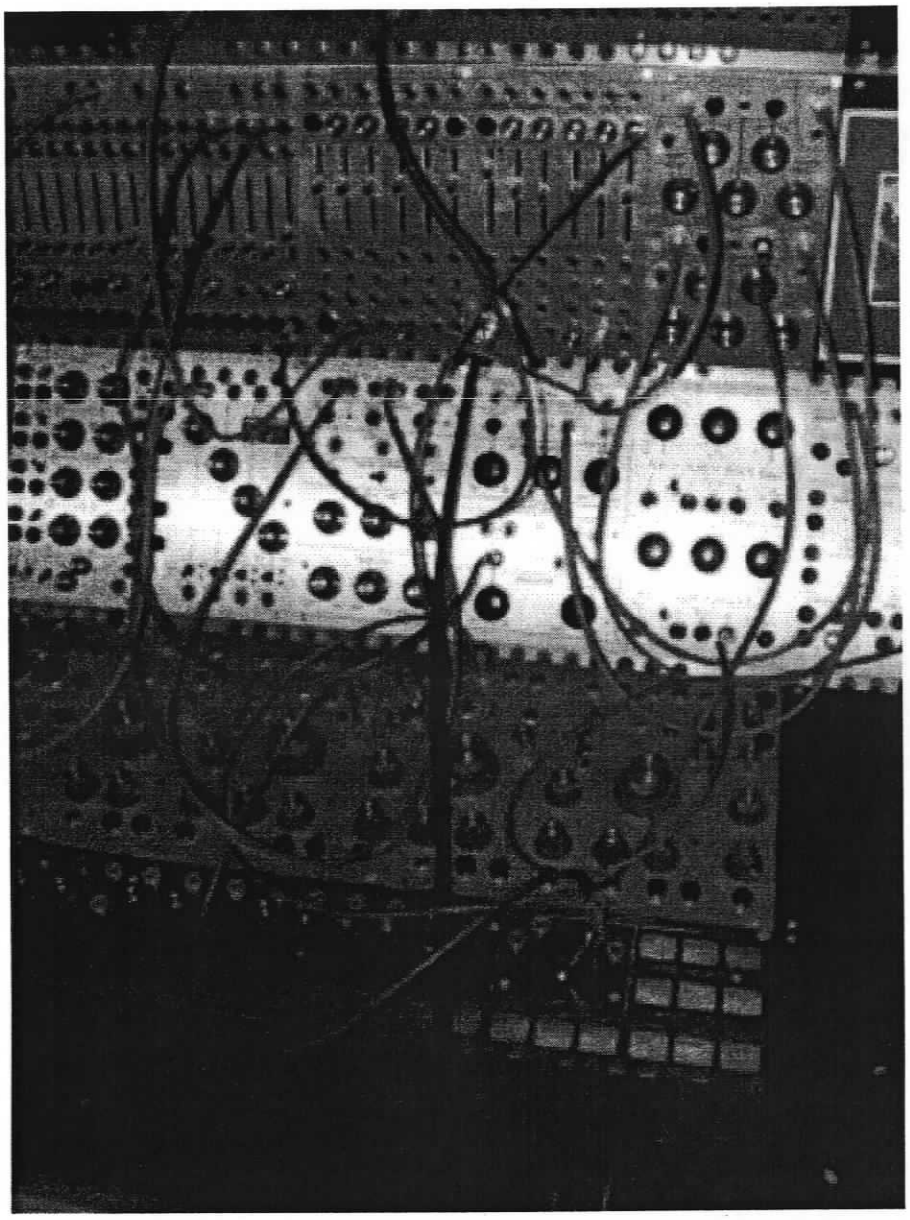

Figure 4.1. *The Programming Interface for an Early Buchla Synthesizer*

flexibility when swapping or modifying the program, once done by some mechanism like changing the Buchla's patch cables, has generally outweighed the loss of contact with underlying mechanism. Besides, anyone capable of programming in a high level language such as C can probably understand the assembly code in short order.

But, the consideration that ease has a relation to utility is the first of two major ingredients greasing the downhill spiral of human-computer interaction. The second ingredient is well known, marketability. The bottom of the pit? Computers that are as easy to use as any well designed doorknob... and about as useful.[5] Programming languages presupposed the proceeding software evolution of desktop applications– the level of indirection at which most of us manipulate the computer most of the time.[6] To use an application, it's not only unnecessary to understand the physical mechanistic causality of the rocks in the machine, it's unnecessary to understand the programming language with which the application was written. The utility of applications can *almost* go without saying and I neither wish to portray them as a petulance or plan to cease using them myself. If the task at hand is pulling up from the database the available seats on a flight to Montreal while you or I wait impatiently at a ticketing counter eight hours jet-lagged, twelve hours since the last real meal and twenty four hours since a shower that wasn't served on a washcloth, the transparency of logic gates and parse trees is pretty irrelevant. But, all these levels of indirection add inconvenience to the gizmo curious– the individualist who enjoys disemboweling their lawnmower engine, dishwasher and stereo equipment. Manufacturers, most notably in the automobile industry, increasingly add hurdles to the do-it yourselfer in the forms of special diagnostic equipment and tools, diverting even simple jobs into service departments of their respective dealers. The indirection should annoy engineering and computer science students who, despite specializing in abstraction, must wade through innumerable lateral inventions before crossing core concepts while some parts remain forever invisible as ac-

[5]Intelligent agents, ambient intelligence, intelligent environments come to mind, eg. a computer that turns on and off the lights in my living room.

[6]Programming languages are the interfaces to applications like compilers and assemblers, ie. applications that enable other applications.

cess to even the source of the programming languages is denied in the name of intellectual property, an odd scenario to place a student. What about the hapless ticketing agent looking for an explanation when the database query fails– times out, gives an error message or results known to be erroneous? Little engines in the spokes? Currents of air and water in the tires? Even if there is nothing the agent can do, a likely and prudent scenario in a distributed, mission critical database application, they are denied the satisfaction of understanding. What's at issue with the evolution from programming languages as the primary application to secondary user applications is the trade of flexibility for simplicity, the obfuscation of physical causality.

And the group that should be most put out are the artists and scientists who in the course of their explorations will need to push computers into areas not necessarily premeditated by distant application engineers. Like a furtive doorknob in a burning building, the lack of visibility of the underlying mechanisms traps the end-user into accepting decisions made by the application designer. Scientists often find an out by simply learning to program, an endeavor more easily accomplished than gurus might like revealed. Numerous languages cater to scientific programming such as Octave, Matlab and, historically, Fortran while usually any language that can handle numbers at the limit of the computer's precision will work fine. The limit of precision? This last constraint demonstrates the importance of better understanding of the machine's underlying mechanism leading to better use– a computer cannot store any number, particularly large numbers or very small numbers such as the difference between two similar "medium" sized numbers, a perhaps surprising realization that results both from the physical design of the circuits as well as the software. Artists may or may not learn to program or for a variety of reasons including inaccessibility to foundation concepts, may explore glitch, ie. using a device for some purpose other than its designers intended [16][17]. Turntablism, the sampling and mixing from vinyl records on the fly, is analog glitch. Or, at least it was before the record player as instrument became so popular that records are pressed specifically for this purpose without the intention that they will never be played start to finish in a linear fashion. Blip and beep electronic music

Figure 4.2. *The wooden Laufmaschine from which the bicycle has descended*

often resorts to glitch such as the intentional scaring of compact disks for the sake of skips. Collage of any sorts, audio or visual, might be thought of as glitch, a recontextualization. Rube Goldberg's cartoons and sculpture certainly employ glitch, as well as wildly complex but mechanistically satisfying design. Duct tape is more often than not employed in a special category of glitch known to engineers as "kludge," performing service where it was not primarily intended and where a better solution exists, such as screws or glue. Transparent design hopes to fuel the same creative motivations driving glitch.

Glitch may often be born of recontextualizing a technology in the absence in understanding of or alienation from its design principles. Hacking, in the older tradition of the word implying something done clandestinely, is sometimes motivated by an alienation from the social structures that support the technology, for example unemployed engineers without inside access to modifying a system, up-and-coming students yet to receive an invite into the corporate fold and the otherwise disenfranchised[18].

Providing access to mechanical casaulity can enable that intellectual urge to understand and ease the process of adapting, innovating and recontextualizing technological artifacts by engineers and artists as the artifact evolves.

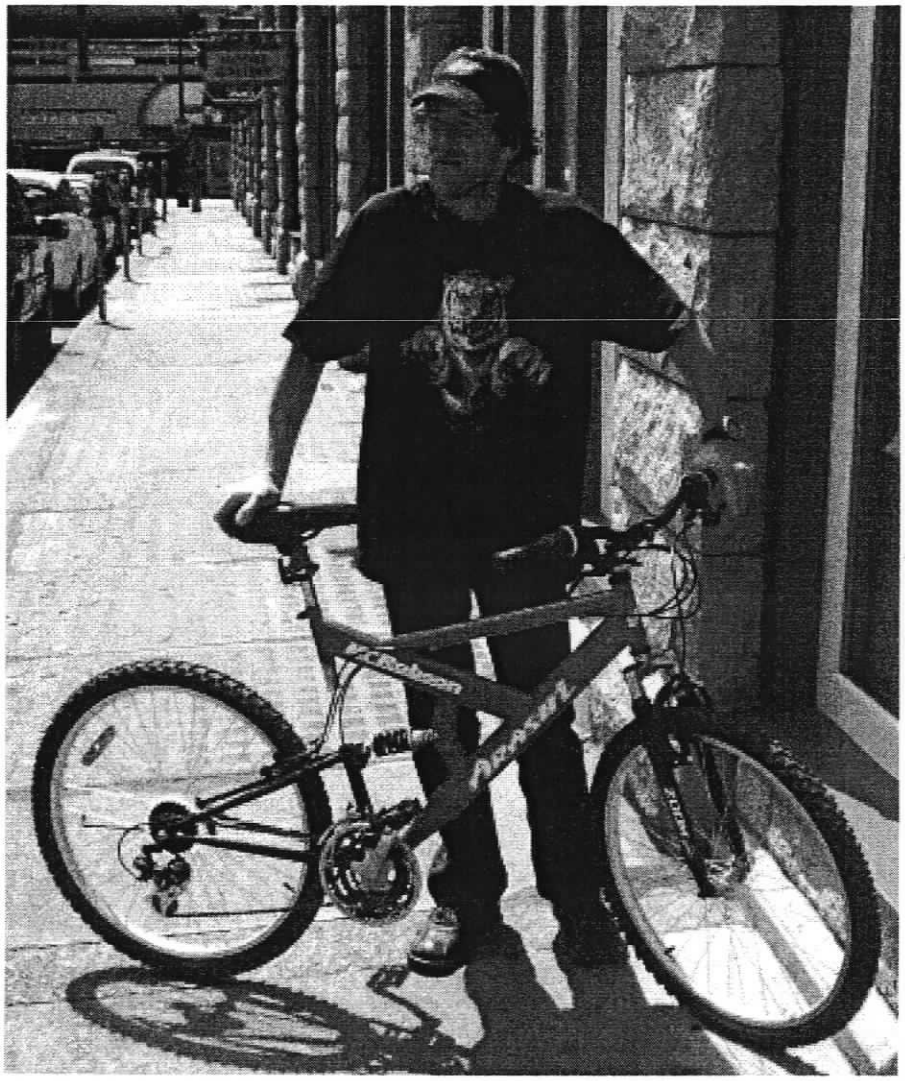

Figure 4.3. *More moving parts– the mechanisms are still mostly exposed*

4.3 Historical Transparency

4.3.1 The Designer's Access to the Turning Points of Ideas

For the moment, the state of the art in computer usability, at least on desktop computers, is dominated by graphical user interfaces and wysiwyg point-and-click environments. These environments were a long time in the making and during their early years, natural and programming languages held most of the attention of HCI conscious software efforts, Table 4.1 [19]. Programming languages showed their early utility to humans and their productivity with their new computing machinery. Furthermore, through the fifties, sixties and most of the seventies, text based computing suited hardware constraints like memory, processing and display limitations. Even tightly constrained applications where a user could only choose from one of several options, it was likely that the choice would be submitted as a text based command, if only a number. As the computer advocates tried to get the machines into more and more hands applying it to more and more tasks, users and programming languages were pushed to find common meeting points. Languages would have to become still easier to use. Possible errors, often arising from misunderstandings of the invisible underlying physical mechanisms, would need to be removed. An infamous example is the memory management devices left in the C programming language, particularly memory allocation, deallocation and pointers. Late binding makes applications more flexible. Initially runtime binding of data to variables was an accomplishment, later late binding was extended to types. Some languages, such as Lisp, more or less hid types from the users altogether. But the user was going to have meet the challenge too. Computer luminaries such as Alan Perlis contended that programming could be a skill for everyone and that computer science would be a core discipline in the liberal arts curriculum [20].[7] Computer

[7] "The writers have been preoccupied with the malaise of man adrift in a wealthy yet culturally oppressive society. Few have studied the real problem– man's decreasing range of influence, the rapid obsolescence of his patiently acquired techniques, and the substitution by technology of vast numbers of trivial choices for the few really critical to his development. Man is not dehumanized or enslaved but he is in danger of becoming

literacy was born. And computer literacy was correctly recognized as an important skill to cultivate in light of the impending ubiquity of the machine. And why shy away from including this new curriculum as a liberal art and not stingily and solely the purview of the specialist? Most core concepts can be covered in a single semester, certainly enough to launch a student on the path of writing programs that model and transform the problems in their respective fields. Computers have met the expectation of mass-market penetration, at least for the affluent, technological sectors. But has the ease of use accompanying icon based compuing come at the price of technological literacy?

Obfuscation has accompanied software interface evolution. The underlying mechanisms, hardware and software, are often hidden from the devices built on top of them. Programming languages move to hide physical mechanism such as memory management. Applications hide the idiom of the programming language with its keywords, elliptical syntax and unforgiving semantics– no matter how usefully flexible they may be. The evolutionary pressure is simplification, a tacit design philosophy that flexibility imbues unmasterable complexity. For added protection, simplification is accompanied by constraints, including the constraint that lower level building blocks should be unreachable.

As mechanical causality is hidden, so is historicity. Compare the images of the laufmaschine and the bicycle, Figure 4.2 [1] and Figure 4.3. While the laufmaschine lacks a drive train– the crank, pedals, gears and chain– the frame and wheels are present. As they are in the contemporary bicycle, available for analysis. Visible and easily inspected, a comparative anatomy of the two machines shows historical development as well as idealistic conservation. In the case of the computer, the elusiveness of historicity and causality are linked in the chronology of a development where older parts still exist, like the bicycle frame, as framework on which the newer parts are built. The newer parts, as already argued, are by their nature often designed with the intention of adding constraints and hiding complexity of the older parts.

irrelevent." – Alan J. Perlis ACM Address, 1963

4.3.2 Early Themes in HCI

The study of human-computer interaction, at least under the broader category of human factors, has been around for nearly a half century, if not longer. The IEEE published "IEEE Transactions on Systems, Man and Cybernetics," generously leaving women out of that dubious formula, between 1988 and 1995. Previously, this publication's topics had been partially captured in "IEEE Transactions on Man-machine Systems," which had until 1967 been known as "IEEE Transactions on Human Factors in Electronics," which until 1973 had been known as "IRE Transactions on Human Factors in Electronics," which dates back to 1960. Many of the bread and butter themes of human-computer interaction are well represented even in those first issues. "Pattern Recognition and Display Characteristics" by W.R. Bush *et al* examines human performance on graphical displays, in this case radar screens [21]. "Computer Languages for Symbolic Manipulation" by Bert F Green Jr. underscores the importance of programming languages to human use of the new machinery [22]. Accessibility opportunities are represented by H. Freiberger *et al* in "Reading Machines for the Blind" [23]. The potential for using natural language in the interface, alluded to earlier, shows up in Thomas Marill's "Automatic Recognition of Speech" [24]. A 33 year young Marvin Minsky submitted a lightly annotated bibliography on Artificial Intelligence, "A Selected Descriptor-Indexed Bibliography to Literature on Artificial Intelligence" [25]. The mouse may have still been some years off, but in the debut volume Richard L. Deininger examines "Desirable Push Button Characteristics" [26]. Even early echos of UML appear in "Operational Sequence Diagrams," a lovely little article diagramming diagrams aiding visualization of more efficient missile destruction, written by none other than Fred A. Brooks Jr, most famous for his book *The Mythical Man Month* [27][28].

Many of the components *directly* evolving into the modern graphical, wysiwyg environment were well developed by 1981. Graphical applications should trace their history at least back to Ivan Sutherland's Sketchpad application developed in 1963 during his PhD work at MIT [29]. Sketchpad used a lightpen to manipulate onscreen vector based drawings. In his introduction to the application, Sutherland proclaimed, "The Sketchpad system,

by eliminating typed statements (except for legends) in favor of line drawings, opens up a new area of man-machine communication." [29] Of course, Sutherland had to type quite a few statements into his TX-2 computer, a computer programmed in an operation code, ie. an assembly language. A few years later, William K. English, Douglas Englebert and Melvyn L. Berman of the Stanford Research Institute evaluated the newly invented mouse along with several other display-selection devices such as joysticks in a series of challenges given to human users [30]. The results were published in 1967 among the pages of *IEEE Transactions on Human Factors in Electronics*.[8] Perhaps most significant to contemporary windowing environments, Xerox PARC's integrated IDE and programming language, Smalltalk, would feature overlapping windows by 1972. In 1981, Xerox would release the Star Workstation featuring WIMP (windows, icons, menus, pointers) wysiwyg desktop. W, the precursor to the X Window System which is now the backbone of most unix windowing desktops, would be written a year later at Stanford by Paul Asante and Brian Reid [31].

While these ideas were decades in the making, or in some cases, decades in the waiting for hardware powerful enough to support software concepts such as overlapping windows. The popular Apple MacIntosh would not be released until 1984. Microsoft announced its windows desktop in 1983, but it wouldn't be until Windows 3.0, released in 1990, that the environment would begin to eclipse the command-line idiom of DOS– I still wrote my undergraduate papers in a DOS based word processor as late as 1994 as I found both Mac and Windows too slow for editing work.

4.3.3 Changing Fashions in HCI

The Association for Computing Machinery (ACM) has been hosting a conference on human-computing interaction since 1981, or as the ACM prefers, "Computer-Human Interaction," CHI conference. The SIGCHI proceedings from the last quarter century provide an informative paper trail revealing the transition in interface paradigms. Computer literacy

[8] In a curious historical example of antique terminology, in English's *et al* article the onscreen cursor is called a "bug" and the process of following its movement "bug tracking."

concomitantly changed; in some cases it picked up a new look such as visual programming languages while in others it seems to have been dumbed down in favor of idioms requiring less background knowledge and practice where context and constraints provide the usage guides to the user, albeit with loss of flexibility. Either way, programming languages became less important, delegated to the domain of the scribes and gurus. SIGCHI papers such as the 1983 submission "What Do Novice Programmers Know about Recursion" or a 1986 report "Does Programming Language Affect the Type of Conceptual Bugs in Beginner's Programs? A Comparison of FPL and Pascal" increasingly gave way to papers like "An empirical comparison of pie vs. linear menus" [32].

A review of paper titles, abstracts where available and, in some cases, the text body itself over the same 25 years of SIGCHI provides a suggestive trace of the change in computer literacy's importance.[9] In my review, themes have been binned into three categories: papers dealing with programming, papers dealing with graphic design issues and, since these topics aren't mutually exclusive, papers dealing with both Table 4.1 [19]. I've made the following interpretive stretches for the sake of better comparing computing environment idioms. The programming theme includes papers on command-lines and command names while the graphic theme includes multimedia. The mixed category is a potpourri including veritable polemic ends of the spectrum; some papers busied themselves with comparisons of the two idioms, sometimes vitriolicly. Other papers more wisely sought synergisms in the approaches such as programming IDEs, visual programming languages and user-interface management systems. The categorizations are my own without benefit of any set-in-stone standard or independent validation. However, to give better insight into my approach and, well, claim a modicum of objectivity, Table 4.2, Table 4.3 and Table 4.4list the themes of each paper more specifically [19].

[9]The ACM SIGCHI conference formally came about in 1983. Prior to that it's immediate antecedents were known as " Proceedings of the 1982 conference on Human factors in computing systems," " Proceedings of the joint conference on Easier and more productive use of computer systems" and " Proceedings of the ACM/SIGGRAPH workshop on User-oriented design of interactive graphics systems" [19].

Table 4.1. *Comparison of Programming themed (P) and Graphic Design related (G) papers for the first 15 years of the ACM's CHI conference along with the most recent year. B are papers with both programming and graphic design themes.*

Year	Total	P	B	G
1981	79	4 (5%)	4 (5%)	2 (2%)
1982	75	13 (17%)	4 (5%)	7 (9%)
1983	59	9 (15%)	3 (5%)	6 (10%)
1985	31	2 (6%)	3 (10%)	5 (16%)
1986	47	3 (6%)	3 (6%)	10 (21%)
1987	46	1 (2%)	4 (9%)	12 (26%)
1988	39	4 (10%)	3 (8%)	6 (15%)
1989	54	1 (2%)	8 (15%)	9 (17%)
1990	47	0 (0%)	11 (23%)	8 (17%)
1991	56	3 (5%)	6 (10%)	17 (30%)
1992	67	2 (3%)	6 (9%)	30 (45%)
1993	70	0 (0%)	4 (6%)	15 (21%)
1994	62	3 (5%)	8 (13%)	15 (24%)
1995	60	2 (3%)	5 (8%)	17 (28%)
1996	55	1 (2%)	2 (4%)	19 (34%)
2005	93	1 (1%)	1 (1%)	17 (18%)

Table 4.2. *Representative themes corresponding to Table 1: 1980 - 1989*

		Examples of Themes
1981	P	error messages
	B	text editors, wysiwig modeling, IDE/shells
	G	visicalc & arcade games
1982	P	indentation & documentation, lifecycles, functional specs
	B	typography, Basic & text editors
	G	menus
1983	P	formal specs & Prolog, query languages, recursion, Pascal tutor
	B	commands & icons
	G	locating items on screen
1985	P	abbreviating command lines, Basic
	B	comparison of text & visual, toolkits
	G	widgets, Phong shading, spread sheets
1986	P	Logo vs. Pascal debugging, semantics
	B	visual programming, visual vs. text debugging
	G	editing, medical cognitive graphics, windows
1987	P	command names
	B	editing commands vs. graphic, data gloves & visual programming, Lisp & IDE
	G	antialiasing & visual performance, windowing, interactive b-splines, CAD, widgets, UIMS
1988	P	command line histories, very simple languages, program comprehension in Pascal/Fortran, documentation
	B	visual programming with Lisp, command lines vs. direct manipulation, animating algorithms
	G	interface graphics, menus, oscilliscopes, voyage of the mimi/educational multimedia
1989	P	command lines
	B	interactive graphics, example-based programming, IDE, UIMS, UIMS/Pascal, visual programming, symbolic math
	G	toolkits, color, UIMS, modeling user interactions

Table 4.3. *Representative themes corresponding to Table 1: 1990 - 1996*

		Examples of Themes
1990	P	
	B	UIMS, layout algorithms, visual programming, visual Unix shell, Smalltalk
	G	widgets, menus, spreadsheets, multimedia navigation, buttons, animation to teach algebra
1991	P	Pascal program comprehension, Unix commands
	B	usability for graphical programming, visual language parsers, Smalltalk, animated algorithms
	G	dynamic icons, animated 3d visualizations, spreadsheets, multimedia authoring
1992	P	COBOL tutor & problem solving, OOD
	B	CAD, 3d design, GUI, UIMS, MAX/IRCAM
	G	multimedia, fisheye views, GUI, widgets, accessibility, CAL, database query widgets
1993	P	Pascal & mental models, Logo media, aiding functions SPSS
	B	animated algorithms, demonstration programming/macros, UIMS, IDE, text vs. graphic queries, visual source code
	G	3d interaction, stereoimages, GUI, menus, whiteboard style UI, internationalization
1994	P	
	B	OOD, IDE, visual OO programming, programmable design environment
	G	text vs. multimedia, GUI for the blind, piemenus, starfield displays, fisheye views, transparent tools, alphaslider
1995	P	psychology of programmers, command aliases
	B	user built widgets, demonstration programming/macros, UIMS, IDE/programmer behavior, Smalltalk/OOD
	G	GUI/online help, menus, tile bars, X Window for low vision, frontpanel for Unix
1996	P	end-user programming of personal agents
	B	demonstration programming/macros, interface for math algorithms
	G	animation/user decisions, CAD, accessibility, PDA/interactive tv, hypertext in GUI, map interfaces, 3d & web, button bars, widgets on small screens, color models, transparent menus, 3d web browser, multimedia CAL

Table 4.4. *Representative themes corresponding to Table 1: 2005*

		Examples of Themes
2005	P	debugging & gender issues
	B	Logo Microworlds
	G	visual search behavior, menus, zooming scroll interfaces, window selection via eye tracking, text on mobile devices, snapping, vacuum widget, toolkits, cursor orientation, thumbnails, fisheye, stencil widgets

The ACM SIGCHI papers reveal the anticlimactic punchline. Interest in programming languages as an HCI opportunity, at least to the preeminent SIGCHI community, was on the decline through the eighties and was all but a trickle after Apple and Microsoft had released the popular windowing environments. Graphic design was taking up the slack in interest. Explaining this trend is a resort to speculation. Perhaps these researchers were on the cutting edge, defining the new idioms that would shortly come to dominate the marketplace and office space. Or, perhaps they were just on the coattails of market success, testing what had already been proven by consumer behavior– simple chasing of research monies. More likely it's both, a mix of zeitgeist and self-fulfilling prophecies born of sufficient hype. More clearly, programming languages did not go away. The new user-interaction solutions depended on programming languages. People still wrote code. But, those people would no longer be the masses or jpfs (just-plain-folks).

Another notion holding researchers' interests before the coming of age of graphics environments was the use of natural languages. A broad category, that includes making programming languages that behave more like the everyday language of person-to-person communications as well as computers that take their commands in the form of verbal instructions. Not to forget that AT&T has made considerable contributions to computer technology such as the C and C++ programming languages and, in part, Unix and that AT&T has long been in the business of relaying voice communications. But the use of natural languages is a logical interface choice given the proficiency almost everyone has with at

least one. That graphical environments took the trophy over natural language processing for bringing the computer to a wider audience may be as simple as that the latter was just ahead of its time. The former was just easier to implement and even it had to wait more than a decade for hardware to catch-reup with the software to the point where users wouldn't frustrate from the wait while the processor labored at redrawing windows . Graphics's beating out of natural language, beit as an aid to programming instructions or verbal commands, further reveals its irony when human graphical acumen is compared with language skills. Consider how many of your friends are even moderately accurate illustrators versus how many can construct a complete sentence. Recall a simple command-line instruction, for instance, how to list the file contents of a directory or how to copy a file. Let's say you selected "cp," the file copy command on Linux/Unix machines. The symbolic representation of the letters "c" and "p," to be fair, involves typography, a graphic design issue, and when entering the command into the keyboard, the font adds a huge constraint on the typography. Independent of this constraint, the glyphs "cp" maintain their meaning across a variety of idiosyncratic reproductions. And, it's easy to remember. Now, without looking, draw the icon for the desktop launcher of your favorite web browser, word processor or other frequently used application. How close is it? Would it be understood by someone else as that icon if passed on a piece of paper at lunch? The icon's use relies on constraint. I recognize my mozilla and firefox icons, neither of which I reproduced particularly well, in part because I also remember where on the tool bar I will find them.

This same experiment was conducted fall of 2005 with a group of eleven undergraduate students drawn from an introductory computer science class as well as an introductory computer music class.[10] Students were given a blank piece of paper and given the following two challenges:

- As accurately as possible, draw a desktop application icon that you frequently, although not necessarily constraining the size
- List any numbers you can think of that a computer cannot represent

[10]University of Victoria Ethics Approval Protocol Number 05-250

In some cases students were also asked to give the approximate number of times the icon was used daily; all students were encouraged to draw a "frequently" used icon. Of course, determining "how accurate" a drawing might be is somewhat of a sloppy business. In this measurement, accuracy is measured by three criteria. First, is there an object missing from the icon? Second, is there an error in positioning, ie. an object drawn in front of a second object when it should be behind, z position determined by overlapping lines of the two objects. Finally, is there a scale error when considering two objects, ie. one is smaller than the second when it should be larger. Each of these criteria require that at least two objects be present. These errors are listed as I, II and III respectively in the results table, Table 4.5.

Of the eleven surveys evaluated[11], three included alphabetic glyphs: twice the 'e' of MS's Internet Explorer application and once the 'W' of MS's Word application. It would be interesting to have sat in on the meetings where these designs were agreed upon and hear the justifications. Here I can only speculate that Microsoft's graphic designers were smartly capitalizing on the well known images of alphabetic glyphs. The errors for the verifiable icons, ie. icons I could subsequently find for comparison, are given Table 4.5.

Of these eight samples, only one was without any of the predefined errors, the application icon for MSN Messenger. Drawing errors were associated with the icons for Internet Explorer, Firefox web browser, MSN Messenger, iTunes, Putty and MS Word.

Certainly this small study doesn't strive to answer cognitive questions of recall or visual memory. In the desktop environment, it's not required to draw the icons, merely select them from a line-up. But consider the results in comparison to typed commands. How often is the letter 'p' mistakenly drawn in mirror image as 'q?' It's unlikely that the frequency of error is anywhere near 7:8, particularly under constraint of the keyboard. Or, how often is the DOS file copy command 'copy' given as 'opyc?' The point here is that while it's easy to have good enough recall of icons and all the student volunteers have no trouble finding

[11] Some icons were not readily available as references when reviewing the surveys and hence are not used in the evaluation.

Table 4.5. *Occurrence of Errors in Student Drawings of Application Icons*

ID	Errors		
	I	II	III
11	X		
10			
9	X		
7		X	
6	X	X	
4		X	
3		X	X
1		X	

and launching their applications, typed or written language is a very familiar idiom already widely used with accuracy by the computing population. While the value of illustration as a basic competency demanded of the population is very valid, its realization is probably no easier than widespread understanding of programming in structured typed-text languages. [12]

A warm, sunny, midsummer Sunday afternoon provides excellent opportunity to pedal my mechanically transparent bicycle for a few hours. I am passed by dozens of antique vehicles, the sort that only come out on such a day. Among the usual convoys of 70s muscle cars, 60s convertibles and 50s landsharks are dozens of cars dating back to the 20s and 30s. A few of these cars, the very ones that passed me today, were built ten to twenty

[12]The second question on this survey, what numbers can a computer not represent, was meant to gage the participants familiarity with the machine's physical constraints. Two responses indicated there are no constraints or none that he or she knew of. Two responses indicated very large numbers and three each believed the computer was incapable of handling irrational or complex numbers. Four responses gave other reasons including an inability to store or represent zero and another singling out Roman numbers. Independent of the validity of these responses, I thought it curious that nobody indicated the computer handled only a discrete set of numbers of limited precision.

years *before* the first programmable, Turing complete (or as close as possible on a machine of finite memory) electronic computers such as the ENIAC, which went online in 1946. The ENIAC wasn't able to store its programs internally, though the idea was out and about at the time, but rather was programmed like the Buchla synthesizer via a rewiring. Unlike the Buchla, the ENIAC was not capable of cool concert performances to the delight of tripped-out groupies. At 27 tons it didn't fold for transport. Stuck in one room, it slowly labored at its hawkish task of calculating ballistic firing tables [33].

The computer awarded the distinction of being the first computer capable of internally storing a program was the EDSAC [33]. It's designer and implementor, Maurice V. Wilkes was awarded the second Turing Award ever given by the ACM in 1967 and is still alive today. In fact, of the first eleven recipients of this most prestigious award, six are still alive today in 2005. All eleven lived to see the release of the Apple MacIntosh in 1984. The first recipient, A.J. Perlis who won the award for work on programming languages and compilers and was quoted earlier as an advocate of general inclusion of computer science in the liberal arts curriculum, died the same year Microsoft released the definitive 3.0 version of its windowing system. Some of these recipients still hold academic positions and are publishing papers such as Harvard professor and 1976 Turing Award winner Michael Rabin who did early work on finite automata and nondeterministic machines and now researched cryptography.

Other players from this narrative also won Turing Awards. Ivan Sutherland of Sketchpad fame won in 1988. Douglas Englebert of mouse fame won in 1997. Frederick P. Brooks, Jr of efficient ballistic missile launch fame won in 1999, although not specifically for that work. Marvin Minsky (1969) and John McCarthy (1971) both had some direct historical influence on the development of Logo programming language through their involvement in AI research at MIT and work on the Lisp language, the parenthetical parent of Logo.

The Turing Award winners are key players in many of the key technologies that define the computer machine we deal with today. And all but a few of them are younger than the

classic cars out trying to pass me during my bicycle sojourn. Beyond making the point that the novelty of even old computer technologies isn't particularly old, none of these earlier pioneers have backgrounds in computer science such as exists in today's universities, reified as departments and degree programs. Of the original eleven award winners over the first ten years, at least five received their final degree in mathematics, three in physics and one in political science. Indeed, Edsgar Dijkstra argued, to the consternation of the software engineering proponents, that real computer science was simply a branch of mathematics in his rant, "The Cruelty of Teaching Computer Science" [34]. What sort of fruit educations in computer science will bare is simply a story that requires patience in its unfolding. However, these early luminaries' training gave them accesses to very low level principles and a fuller picture of the machine, still naked in its mechanistic causality.

4.4 Literacy with Machines, Literacy of Machines

Presumably, with a field as new as computer science, many of these researchers saw their efforts as the incipient baby steps. "The best way to predict the future is to invent it," said 2003 Turing Award winner Alan Kay who in 2004 gave a keynote address to the Object-Oriented Programming, Systems, Languages and Application conference in Vancouver,BC titled "The computer revolution hasn't happened yet." Kay is likely most famous for his Smalltalk language written at Xerox PARC in the early 1970s along with help from colleagues Dan Ingalls and Adele Goldberg among others. Kay's vision reflects those of Perlis, Minsky and Papert in regarding computer science as a core curriculum of general value to the population at large and that the population at large should have access to a level of control of the machine that employs its full flexibility for modeling, simulating and experimenting.

> The "trick," and I think that this is what liberal arts education is supposed to be about, is to get fluent and deep while building relationships with other fluent deep knowledge. Our society has lowered its aims so far that it is happy with

> "increases in scores" without daring to inquire whether any important threshold has been crossed. Being able to read a warning on a pill bottle or write about a summer vacation is not literacy and our society should not treat it so. Literacy, for example is being able to fluently read and follow the 50 page argument in Paine's Common Sense and being able (and happy) to fluently write a critique or defense of it. Another kind of 20th century literacy is being able to hear about a new fatal contagious incurable disease and instantly know that a disastrous exponential relationship holds and early action is of the highest priority. Another kind of literacy would take citizens to their personal computers where they can fluently and without pain build a systems simulation of the disease to use as a comparison against further information. At the liberal arts level we would expect that connections between each of the fluencies would form truly powerful metaphors for considering ideas in light of others.

Written in his essay, "An Early History of Smalltalk," Kay touches on some of these beliefs motivating the design of Smalltalk [35]. Smalltalk, in addition to containing a design brilliance that makes it one of the most influential pieces of software on today's desktops, contains numerous ironies. Smalltalk was one of the first object-oriented language and still one of the very few fully object-oriented languages, ie. even primitives like numbers are objects. The inspiration for classes and objects came less from the familiar text book examples of car is to vehicle/bus is to vehicle or worker is to employee/boss is to employee examples as from the idea of biological cells. Each cell is selfcontained with its own states and machinery for accomplishing given tasks, but cells' designs are based on reusable and hierarchical patterns. The design reasoning was that such units would ease the sort of modeling-programming tasks that literate 20th century citizens might find themselves doing. Object-oriented engineering has somehow slipped out of the hands of the masses, morphing into an industry of specialists. Thick tomes on Design Patterns and Object-oriented Systems Engineering make up intermediate to upperlevel study for computer science students [36]. Companies hire special architects explicitly and solely tasked

with identifying these structures of programming language components– structures used less frequently in ad hoc modeling than inside megalithic applications run on IBM mainframe computers in out of the way corners of heavy industries like freight shipping, a not unlikely environment to find IBM's Smalltalk-80 implementation.

But, Smalltalk was not simply a programming language. Among many novel features was its innovative IDE. The development environment that aided in the manipulation of the objects provides frameworks for the programmer to create classes from which those objects take their behaviors and did so in the context of resizable, draggable, overlapping windows– one of the earliest such environments and most clearly the model for today's windowing systems.

In Smalltalk we find a synergism of the programming, command mode approaches to interaction and the context aided, wysiwyg, point and click graphical environment. Unfortunately, what has been retained on the desktop computer over the years are the easiest to use components, the windowing environment. The implementation language itself hidden from most end users. The part of the equation demanding effort by the user has been dropped by the major vendors like Apple and Microsoft, mechanism again hidden away. In one way, Smalltalk added a level of indirection between the program and the machinery below it. The language and its environment run on a virtual machine (vm), ie. the program instructions tell the vm what to do. The vm must then translate that to instructions affecting the underlying computer equipment, like reads on the disk drive. The drawback is further removal from mechanistic causality. One benefit is portability. However, Smalltalk makes up for this indirection with a different sort of transparency. The language and environment itself is written in Smalltalk and is accessible to the programmer-user via a class browser that shows both software design relations and the underlying logic code. In fact, programming is really just a process of making extensions to the virtual machine. The code driving the virtual machine is not only open source, but it is conveniently organized. It's available to view and a source for understanding the design and syntactical idiom of the language. The programming problem can be approached in either a constructionist or deconstruction-

ist direction. Building models is facilitated by modular components while deconstructing where the goal to understanding is, at least, a possibility via accessible source– a possibility that reciprocally aids in the construction process by providing patterns and explanations of the building blocks.

Kay provided a mechanism for those who wanted to "predict the future by inventing it" in allowing the dissection of the present. If, metaphorically, the technologies and ideas encapsulated in their designs are viewed hierarchically, then like the physicists and electrical engineers of the early and hence lower-level computer technology, innovation will be the option of those who access and take time to understand the building blocks and how they can be reconfigured, or the artist who recontextualizes by glitch. When constraints are defined by forced ignorance via a stinginess with the details, users may relegated to follow. Glitch provides additional sublimity in its subversiveness; it breaks the constraints. But with the exception of trivial or highly prototypical artifacts, it's not free from performance rigorousness– quoting out of context may be the fear of many writers, particularly those who touch sensitive topics and, perhaps because of this, is rightly looked down upon as an intellectual *faux pas*. Using technology out of context, like a chainsaw to trim a beard or duct tape to secure an airplane wing, demands a certain level of responsibility on the part of the improvising inventor– a responsibility that, again, may be better met if one understands the underlying units and their failing points, ie. even glitch can benefit from an ability to deconsruct to some arbitrary level without destruction.

4.5 Performance, Good Magic Tricks and Transparency

Earlier I touted the transparency of mechanical causality in many musical instrument interfaces, such as a piano or guitar. Sound, as any radio will demonstrate, can be free of visibly demonstrated. Just think of the "distant sound" of this or that "dancing down the twilit street." The orchestra sitting in the pit at an opera or ballet defies visual analysis as does recorded music played back via compact disc or similar device [37]. Perhaps many

of us are indifferent to this slight of hand with our focus on the results not the means– at least until we become interested in reproducing the results. Or, perhaps the disconnect from causality is less removed than this analogy suggests. We may be familiar with operatings of an orchestra from previous experience. Furthermore music has traditionally been filled with reference to human scale. The length of a bowed tone has a relation to an arm; the attack, loudness and duration of a horn to the size and force generating capacity of a lung; a rapidly played sequence on the piano to movement of the fingers and drums, back to the arms again– no matter how fast the roll, the arms remain a limiting factor. Our bodies may differ, but within the variance, proprioception provides an intimately known scale, a scale reflected in music.

Newer technologies offer other disconnects. Sampling and the collaging of samples via sound editors on laptops provides a cited example of mechanistic disconnect [37]. In some genres, such as turntabalism, cause and effect is still very much alive and delivered with considerable panache, sonically and visibly. While the cause of each timbre and pitch of the samples has been lost in the immediate context, the dynamic impositions unique to the work are lavishly on display. Synthesizers may use familiar keyboard interfaces, capitalizing on the familiarity to piano players, but the timbres generated are not so easily related to a source like piano strings. Laptop music often includes both approaches. The laptop may be used to synthesize new sounds or playback samples. With the audience probably familiar with both possibilities, the performer, busily typing away could just as easily be checking email as improvising live music. At a recent concert I attended, a band used a combination of approaches including sampled music and live analog instruments, each musician playing numerous instruments, a process that sometimes involved setting the instrument with a riff to loop until inactivated. The inclusion of a vocal track was apparently one too many components for them to incorporate on the fly. So a prerecorded vocal track was unabashedly used without even the pretension of lipsyncing. Beyond any value judgment on such a performance, the vocal track ended up dictating the tempo and key of the music and lessened the opportunity for improvisational interpretation of environment like

audience enthusiasm, a distant trainhorn or the hapless waiter who just spilled five pints of beer somewhere between the dancers and wallflowers. Motivations for attending a live performance may include many social factors only tangentially related to music– the warm chaotic crowd, flirtation, alcohol, etc...– for which many ostensibly pay to see live music. The possibilities of being the rubes to precorded showmanship is disconcerting. Even with an accepting audience, as many laptop fans clearly must be and with music capable of holding its own with the listeners' interest, performers are aware that the revealing of cause and effect can enhance the satisfaction. Juggling flaming pins is that much better and I can't recall a drum solo that became less exciting when a piece of broken stick went flying across the stage. Magic amazes with its slight of hand. But truly great tricks remain so even after their mechanism is revealed.

Visibility, a good conceptual model, mappings and feedback are solid notions to consider in design work. Mechanistic transparency, the accessibility the physical components of causality, should be appended to that list. In an ideal design, mechanical causality and historicity are, if not blatant and visible, at least findable, traceable and dissectable. While neither the machinery nor its history may be simple, loopholes in constraints liberates the user. Full flexibility of the computer can be restored, the satisfaction of understanding redeemed and the opportunity to innovate made clear.

Chapter 5

Flowers for Algorithm

5.1 Preface

Perhaps a lingering and nagging question remains: why slow oneself down with unnecessary details of how a technology works? Afterall, isn't that the definition of mired? The proceeding essay essentially boils down to an argument that there's value in rolling one's sleeves up and getting involved, making an attempt for an intellectual and visceral understanding of these mass-produced, hightech artifacts. Roles for intuition in understanding are discussed along with roles for the senses in developing intuition– basically, making another philosophical case for hands-on learning. In keeping with the previous theme, I advocate a design strategy for common artifacts like computers that enables that sort of direct involvement.

The essay, however, is not expository– its form, as with all essays, an important part of its content.

5.2 Essay

cell phone Bob
in the cell phone mob
talking madly to himself
of how he rolled out of bed
and landed straight on his head
but hadn't found the floor

It's not untypical to start an essay with a quote from some notable, distinguished author. They may or not have worked within the discipline of the ensuing text; noteworthy analogies from other disciplines portend to underscore the universality of the underlying pattern, the essence of the proceeding topic. Quotables from thinkers long dead, preferably thousands of years dead, add gravity, the weight of so much death, advice given from one as omnipotent as a ghost. The filter of time has resulted in the impression that there were fewer, and subsequently loftier, thinkers in antiquity. The ancient quote provides the root node of a hierarchy upon which the novel argument will be based. Or, perhaps it simply demonstrates the insight of the quoted, their ability to envision modernity from such a long way back into antiquity.

The use of the quote is literally a literary collaging technique. Musicians are often said to "quote" when playing a bit of borrowed melody, perhaps during an improvisation. This latter use of "quote" is perhaps less similar to the quotation practiced in writing than sampling, such as done for a musical synthesizer or by a tape loop composer or turntabilist– there's no avoiding recontextualizing the quote, whether it's used to reinforce an argument, provide counterpoint or launch into a refuting diatribe. There's no avoiding borrowing some of the timbre of the quote, using the color provided by the originator. It may provide historical evocation, a temporal transplanting. The sampler may highlight their splicing bravado through an improvement in the dynamics of the original, otherwise cheeseball, recording. Art wishing to prove itself by magnitude of emotional response may use unex-

pected juxtaposition of quotes to create the uncanny or humorous– uncanny and humorous as in a sample off a scratchy vinyl recording used by the band Man or Astroman which I now quote, "in case of nuclear attack, the preservation of records is vital if this country is to maintain its economy and carry on its way of life" [38]. This essay starts with an original poem, unique to this essay. The first paragraph provides a brief critique of essays that commence similarly but with a collaged bit of quoted text.

Quoting out of context may be an error, malicious or glitch, ie. intentional recontextualizing. The author's intended meaning may exist so contentiously or obscurely that little hope exists to find a usage agreeable to everyone. For instance, Friedrich Nietzsche's writing is one frequently cited and cited as being frequently cited out of context. Prudence suggests avoiding conjuring such a writer, but his ideas are presented so forcefully and eloquently many can't resist. And how can we blame his fans and detractors from their errors when he himself employed a style that often included apposing comments within the same passage for the sake of contrast and an enthusiastic urge to revise one's argument in realization that, over a lifetime, contradictory claims would surely emerge– the presenting of an immutable edifice of achievement bowing out to the sublime demonstration of growth and reflection. This essay quotes from the style of *The Gay Science*, which opens with a prelude in verse [39].

> A "scientific" interpretation of the world, as you understand it, might therefore still be one of the *most stupid* of all possible interpretations of the world, meaning that it would be one of the poorest in meaning. This thought is intended for the ears and consciences of our mechanists who nowadays like to pass for philosophers and insist that mechanics is the doctrine of the first and last laws on which all existence must be based as on the ground floor. But an essentially mechanical world would be a *meaningless world*. Assuming that one estimated the *value* of a piece of music according to how much of it could be counted, calculated, and expressed in formulas: how absurd would such a "scientific" estimation of music be! What would one have comprehended,

understood, grasped of it? Nothing, really nothing of what is "music" in it!

And what would one have understood in "formulas" if they are known as nothing more than mechanics? In addressing this question, a thorny issue hanging from the previous essay may be addressed. I've argued for transparency in design of artifacts– not that components should be rendered invisible but that they shouldn't obscure their mechanical relation with other components. What's the value in this clarity? Given the mechanical complexity of many of our day to day artifacts like computers or over-the-counter drugs, isn't it too much to demand that the user become intimate with workings of their design? Perhaps. The urging here is not that the user must make this effort, but that the design avoid putting up unnecessary hurdles to those who do. The artist who examines the mechanical may indeed find value in the result of meaning. The remainder of this essay will be devoted to a nexus of mechanics, meaning and artist; specifically, how access to mechanism allows the artist to construct meanings, in the process mollifying obsequiousness to a machine's fabricator. The outline will start with an examination of causality and a justification based on intuition then proceed to discuss correctness, synthesis and knowledge. The bigger picture remains a justification for literacy, in this case literacy as practiced with a common mechanical appliance, the computer.

In a few cases, formulas comprise an indispensable component of mechanics where it's unnecessary to even make much of a distinction between the two. Formal mathematical analysis, the guardian of the formula, wouldn't be much without the formula; the mechanics of the enterprise organically includes the formula. But, pencils, conferences, cocktail parties and watercooler debates are also part of that mechanics, particularly when considering how the formulas come to have a meaning, ie. that they are true or are not [40]. Computer programs are formulas and integral parts of the computer's mechanics. The actual rendering of the formula, ie. text typed out on the screen and the associated underlying data structures in the logic circuits and memory devices, is vital to mechanics of the task. Without the program, the mechanics breakdown. Take the formula away, and the computer is as useful as bicycle without wheels. The equation modeling the flow rate from a cask of

wine exists independently of the cask and carafe. The wine will flow without it. The design is straight forward and familiar enough that to increase flow, most of us will either try to increase the size of the hole or increase the pressure by tilting the vessel; although, perhaps not with a large wine cask. Many besoted partiers without a single day of calculus to their credit know to pump the keg, thereby increasing pressure and flow. These are formula-free understandings of mechanical causality. Formula-free, formulated or modeled by formulas, each mechanism entails causal expectations.

Causality. One thing proceeding to the next. In physical systems something causes something else thus suggesting a temporal relation. First this then that. Gear one turns gear two. I ate bad fish then I got sick. The notion of causality exists temporally in these exams. My desk holds up the computer, lamp and stapler. The support against gravity is instantaneous and continuous and free to ignore time from moment to moment as much as any of us are able to ignore time. A mathematical proof is temporal in that there is a series of steps; although, the steps can generally be easily run in either direction and, indeed, proofs are often arrived at by knowing the start and the final outcome, which is assumed to be correct, and subsequently filling in the rest. The causality exists between the steps, like a thin film existing between gear one and gear two. It is the stuff between the steps that causality is made of. And in the case of proofs the stuff that allows the steps are the axioms, the basis of the argument, the rules allowing the proof to operate mechanistically. The axioms, in turn, may very well have once been theorems, algorithms, programs in need of proving but that have since become widely accepted as an acceptable starting point.

A dilemma of magnitude should quickly be apparent particularly when the process is considered recursively. The axioms needed proving; the axioms for those antecedent proofs needed proving and so on down the line. If it wasn't already, causality suddenly seems pretty hard to pin down. The thin film between the gears, holding up the lamp and justifying the next line of code becomes an infinitum of even thinner layers. Mathematicians, thinkers who wade regularly into such patterns, have among their ranks some comments.

> On the contrary, I find nothing in logistic for the discoverer but shackles. It

> does not help us at all in the direction of conciseness, far from it; and if it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?

Richard A. De Millo and his colleagues attribute this bit of reassuring reflection to the mathematician Henri Poincaré in their article without citing a specific reference [40]. It's a curious comment for a worker who preferred to work his proofs from first principles, from the basic to the complex. A comment of frustration perhaps? Most of us are willing to jump blindly into the middle, proceed without full understanding. Indeed there may be no other way to tackle many problems. Their magnitude is unknown until they have been circumnavigated; the first principle wouldn't even be recognized until its progeny understood as they can at a local level. Indeed, the entire question of which principle should be first principle might be asked. While A may exist independently of B and B not independently of A, in the process of discovery, B may be more distant and even unreachable without passing through A just as I cannot reach the basement of a house without passing through the first floor held up by the basement.

Few would doubt the utility of formulas. Most would prefer that they operate correctly, particularly when critical to health, happiness and well being. But the question of how well they can capture truth is less clear, a murky debate ripe for philosophical pondering and rich in history of such musing. The debate flared up in the computer field, not coincidentally, about at the height of interest in computer literacy and during the incipient rumblings over the new discipline of software engineering. As a consumer, I would certainly hope that engineers take what precautions they can to ensure that formulas are correct. But programs are very complicated formulas often working in highly dimensional space with data that's difficult to impossible to predict.

One school of thought is forcefully argued in Edsgar Dijkstra's reflection "On the Cruelty of Really Teaching Computer Science" [34]. Dijkstra makes a strong case for the teaching of abstraction in computer science at the expense of software engineering, which

he calls the "doomed discipline."[1] The debate parallels the unnecessarily recrudescing struggle between rationalism and empiricism, or at least their operators deductive logic and induction. In this view the computer is a symbol manipulator, nothing more. Programs are formulas and like their mathematical counterparts, can be proven. The can be proven using formal methods and Dijkstra argues they should be proven, making formal proofs a cornerstone of computer science education. Of low regard in Dijkstra's argument is the more empirical approach of engineering, ie. model and test. Testing in engineering may include stressing the application against possible input data. But Dijkstra would deny students the feedback of running their programs, avoiding the iterative approach of coding, testing, modifying. Experimentation is removed from the programmer's toolbox.

> Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding[34].

Dijkstra suggests a joy that is beyond the engineering sureness of a correctly working program– Dijkstra appears to want to clean up engineering, for instance, suggesting that programming bugs should be called by the more appropriate name "errors." But, the motivation appears from a different place than wanting to build better bridges or airplanes. The formal proof is joyful because of its allegiance to truth, and truth must be deductively knowable, ie. rational. Of course, the formal proof is true in that it follows its rules or, alternatively, contains syntactical errors and does not. The proof's correlation to other truth remains an outstanding dilemma as does meaning, except, perhaps the meaning that the universe operates on causality quite deeply. The enemy of the formal proof in this argument is intuition. But in this case the enemy is left ambiguously defined as the ambiguous, that which may interfere with formal proof's clarity. Seymour Papert describes this a misvaluing of intuition by unfairly making it the scapegoat of our mental errors. Indeed, intuition should be defended for its valuable role to mathematics and thinking in general.

[1] Dijkstra also harshly criticizes artificial intelligence. The argument, simply put, is that computer science wastes its energy attempting to mimic such an inferior device as the human mind. It should be striving to offer a better alternative.

DeMillo, Lipton and Perlis explicitly identify themselves as "antiformalists" in their previously mentioned article, "Social Processes and Proofs of Theorems and Programs," a broadly viewed but deeply implicated pondering that predates Dijkstra's "On the Cruelty" lecture by a decade. Ostensibly, the goal is a defense of software engineering, an empirical approach of trial and error striving to write the best programs possible to deal with messy "real world" problems. The problem could be simply stated as one of economy, and most of us would probably be satisfied. Who wants to prove the program tracking payroll for the French National Railroad, which in 1979, apparently had 3000 different pay rates differing, for factors as esoteric as the grade of track on which the train had traveled [40]? Or, from my own experiences, how to move Sumitomo Pharmaceutical's GATC gene expression data and its correlating meta-data of messy patient medical histories from spreadsheets into a relational database– who smoked, who has an allergy to shag carpeting, who had a hysterectomy in 1976. When theorems can go thousands of years unsolved, or appear solved only to be "proved" incorrect hundreds of years later, good enough is good enough for programming everyday problems, let the trains run on time, mostly if not always. These authors take a further step in contradicting the position of program specification as an exercise in formal mathematics by examining how proofs come to be believed, a process they find rich in its social dimensions and more often than not bystepping that immutable gluing film of causality.

While the social process, itself, may have mechanism and causality, hypothetically revealable by sufficient dissection, a key philosophical difference is the many substitutions of "is" with "believe." The epistemological question addressed by rationalism, empiricism and its many reasonable synergisms is not one of how we come to know truth but how we come to have beliefs about truth. At issue is the relationship of what we know with what is actually going. It's a reasonable difference since, no matter how direct the connection, the inside of my head and the inside of the sun are two different places, or at least there's no reason not to maintain that they are. Now, it's important to pause for a moment. Since many heady philosophical issues suddenly present themselves. These questions have been better

dealt with elsewhere by more capable philosophers. The task presently is to raise questions of design of software and the use of the computer. These philosophical inquiries will offer context for considering humans interacting with their artifacts, computer software in this case. Differentiating between what I know about the inside of the sun and the actual inside of the sun admits the possibility that mental models are a reasonable analogy to thoughts. "Mental models," afterall, rolls off the tongue so nicely it's easily accepted as euphemism for thinking. But, in some paragraphs further on, "mental models" may become a bit stifling. "Mental" suggests confinement to the head, an unfortunate limitation that ignores not only the role of the rest of the body in thinking but ignores the role of artifacts in thinking as well. "Model" underscores that the thought is only representative of some other real thing. While this shouldn't overtly hurt the subsequent argument, it may belittle the very real connections between the real and the real model. Back to belief versus truth. Deduction, when following its own axioms, certainly can be proved more or less correctly. Not every math problem ever solved, say on student tests, has been solved correctly. Transcription errors and logic errors abound and even by their own internal rules formal proofs can go astray. What we know, well, just isn't always correct. So deduction isn't perfect– this a realization before even discussing deduction's reliance on messy old empiricism. Deduction isn't perfect in obtaining the correct answer, but can deduction know perfectly? Can I answer any possible question about the internals of the sun without a mental model that's a perfect little sun burning in my head? Even such a complete bundle of thermonuclear thought would be incorrect in magnitude.[2] Of course, most logicians, including Dijkstra I believe, would argue that few proofs are worked out by finding all combinations of possibilities [34]. It's quicker to work with definitions than every member in the set of the definition. And with

[2]Without specifically referencing mental models, Emily Dickinson optimistically noted the substantial carrying capacity of the brain [41], "The Brain is wider than the sky, for, put them side by side, the one the other will include with ease, and you beside." While I agree with the ecumenical punchline of Dickinson's poem so far as god as mental construct, "The brain is just the weight of God, for, lift them, pound for pound, and they will differ, if they do, as syllable from sound," the first verse suggests an impossible omniscience, an ill-fitting one-to-one correlation of mind and sky, a sky without horizons.

the introduction of definitions, rationalism, the prodigal explanation, has been reunited with empiricism. Thinking deductively about the sun starts with the observation of the sun or at least a suggestion that the sun exists such as a midwinter postcard from the tropics. On top of that, there's nothing stopping me from examining my own rational thoughts empirically since, like I sense the sun, I can sense my own thoughts.

Writing a proof may be accomplished with paper and pencil. Believing a proof is entirely different. The social process of which DeMillo and team write occurs at conferences, cocktail parties, lunches, watercooler chats among other forums that include both scholarly and informal discussions. The ground turned out by the tines of this process is belief about the correctness of the proof. That "proof" needs proving even in its completely explicated form should raise doubts about the truth-stuff gluing together the immutable steps to the proof. That the social processes determine belief in a theorem and not necessarily the formal steps of the proof, which may not even exist in some cases, might be demonstrated anthropologically. But, DeMillo, Lipton and Perlis, being computer scientists, continue their analysis with a critique of the formal proof. Where Dijkstra finds the magnitude of the computing machinery "radical" and "revolutionary," they find the magnitude of proofs monstrously debilitating– citing, as an example, Alferd Whitehead and Bertrand Russel's lengthy *Principia Mathematica* . While that three volume work covered many basic mathematical axioms, it hadn't even taken on the three thousand pay rates of the French National Railroad. Historically, once accepted proofs can come to be discredited. Famous examples include Fermat's Last Theorem and the Riemann Hypothesis that have seen proofs come and go [40]. Similarly, DeMillo *et al.* describe an incident with two proofs derived by separate research groups working in the area of homotopy, the continuous (or analog) transforming of one function into another, which incidentally was also a key interest of Poincaré. The two proofs contradicted each other. But when the research groups exchanged results, neither could find fault with their peer's work. The successful proof can be equally confounding, DeMillo *et al.* noting that Paul Cohen's work with forcing in set theory was hardly understood even by competent mathematicians when it first

appeared. Finally, this assumes that people can stomach, without erring themselves, the dense notations often employed in formal proofs.

These problems all share a common thorn in the finger of truth by formal logic; humans must experience and interpret the proofs. There is a transformation from the original informal specification motivating the proof to the actual proof. While the proof may be deductive, it must be sensed, experienced and very likely only in small parts at any given time. One cannot focus on the entire proof simultaneously, most of the logic being held in place by memory but beyond the horizon of immediacy. That all might even be reduced to a proof has, itself, yet to be proven. The possibility exists on top of a faith, perhaps probabilistically based, ie. that since some things work out as proofs, it's logical that everything should work out as a proof– induction justifying the admissibility of proof in truth-saying. It's probably a safe conjecture that everyone I've dragged into this debate, including myself, *likes* formal proofs to some degree. After all, these are the musings of mathematicians and computer scientists each of whom has found utility in the formal proof. But the mechanics are not always similar, ie. formal proof versus experimentation. What is understood as the meaning of the formula articulated as "you don't get something for nothing?" The correctness of the formal proof may be reassuring, free of vagaries and ephemera, but such confidence comes at the price of ignoring the question of causality by predefining acceptable increments to proof's movement. The other position, for the moment labeled the antiformalist, casts suspicion on all that is sure by not only questioning the origins of first principles but also the possibility of ever finding all first principles and connecting them to all other first principles, whether logically impossible or merely humanly impossible. Suspicious, but not necessarily uncanny. One needn't prove gravity to get out of bed. The antiformalist can boast strong support in our deep experience of the many things that work most of the time as we expect them too– that and an approachable notation.

The two opinions could be distinguished by the admissibility of intuition. The formalist regards intuition as intuitively misleading, a crutch of laziness where the rigor of proving has yet to extend. The Dijkstra human must be truly lost in the world with their inherently

faulty intuition. Curriculums stressing formal proof should be designed "to further sever the links to intuition" [34] But this hardline stance is comical. Intuition is ineluctable. It is intuition that would drive a belief that formulating the absolute is possible. Even if all was proved, formally, the exegesis would be so enormously beyond the human attention span, intuition would provide the compass to navigate the argument. And, this is a big punchline, intuition *is* the glue holding together the components of the proof. The validity of a causal relationship will only be broken down so far, irregardless of whether further reduction is possible. The process needn't continue beyond the point where intuition has been satisfied. This last point essentially provides a defining description of intuition, the unit of proof that is accepted as obvious.

As an aside comparing optimistic and disparaging views on intuition, let's return to Poincaré for a moment, a mathematician well known for rigorous proofs and previously noted in this essay for recognizing the role of the aesthetic in mathematical research. In the chapter of *The Value of Science* titled "Intuition and Logic," Poincaré gives intuition a dual role in mathematical inquiry in some ways similar to phenomenological suggestion of its meaning, ie. he appears to regard it as a basic unit of truth [42]. But he finds these units differ between different individuals who he calls "analysts" and "geometers"

> M. Méray wants to prove that a binomial equation always has a root, or, in ordinary words, that an angle may always be subdivided. If there is any truth that we think we know by direct intuition, it is this. Who could doubt that an angle may always be divided into a number of equal parts?

Poincaré goes on to point out that Méray, an analyst, does in fact doubt the truth based on intuition, going on to develop a several page proof to demonstrate to himself that it is so. Professor Klein, "the celebrated German geometer," feels no such need and instead relies on analogy to electrical currents on metal surfaces to develop his justifications. Poincaré doesn't fault intuition for mistakes but rather blames imagination, for instance, we cannot imagine a line without width, though that is indeed the part of the definition of a line. Furthermore, the two cohorts differ in their use of sense, the geometers being "intuition-

alists" with an urge to paint and the analysts "logicians," who also rely on intuition but one free of sense and rather an intuition of pure number. In this way, Poincaré presents an argument between those compared above, that the geometers are working more like Papert while the analyst more like Dijkstra. The roots for this schism, he posits, are deep, "The mathematician is born, not made, and it seems he is born a geometer or an analyst" [42]. The differences are fundamental and ineluctable properties of the human mind, a position which will not be further debated here other to say that this appears, to me, far too self-fulfilling a prophecy, one that fails to account for the plasticity of the brain or that analysis and geometry are social and cultural enterprises that occur outside an individual as much as within and perhaps needn't exist at all. Stopping short of speculating any value judgment Poincaré might have made on the utility of intuition, if one were to believe a mind's predetermination to a type of intuition is ineluctable (though I do not necessarily believe this), then surely intuition plays an invaluable role in human experience, understanding and construction of knowledge and culture, its influence and importance equally ineluctable.

A former employer of mine, Dr. Alan S. Segal, more than once chided me "assume makes an 'ass' out of 'u' and 'me'." Of course, his research career, in small part, rested on the outcomes of the electrophysiological experiments I had been hired to conduct. His concern was understandable and softened by good humor and tolerable criticism. Intuition, to be fair, has no perfect track record. It gets us lost down the wrong roads, the wrong lovers, the wrong fish taco, the wrong shirt-pants ensemble and the wrong lines of reasoning. But, these misses provide testament to intuition's ubiquity, not an unavoidability of abject council. Intuition can join the ranks of deduction, formal proofs, empiricism and any other prescriptive or descriptive model of thought– a "bug" is an "error" and by any other name would be equally incorrect. But to prelude the next two themes, intuition is trainable and intuition is more than gossamer. Intuition can supply as firm a grounding as any available.

Computer science's cousin cognitive science has, like philosophy, endeavored to provide a more meaningful understanding of intuition than as faulty faculty and scapegoat to

our poorer decisions. At issue, for most, is not the existence of intuition or even that it plays a role in mathematical understanding, but its trustworthiness and utility. This juncture returns to the topic to computer literacy. One of the earliest proponents of teaching computer programming, particularly to a receptive audience such as children, was Seymour Papert– co-founder of MIT's artificial intelligence program and careful student of Jean Piaget. Papert's contemporary, Alan Kay, suggested literacy was not the ability to read marketing labels but the fluency with which to plow through philosophical writings and prepare fluent critiques; in closer reference to mathematics, literacy was the ability to identify a problem that benefited from mathematical modeling and possession of the necessary skills to accomplish that task, skills that now invariably involve computer programming of some sort [35]. Arguably, these are richer goals than the more commonly heralded expectations of operating an employer's office productivity and database software. Papert elevated the bar higher yet. **First, debugging (or "de-erroring" as Dijkstra might prefer) is a widely applicable problem solving skill that transcends disciplines and is easily honed in the programming environment.** If there's a single sentence in my thesis looking to make the case for computer literacy, it is this. That debugging computer programs involves problem solving is unlikely to be contentious. The argument values programming for reasons other than simply operating the computer; thus claiming programming to be extraneous to the masses when more user-friendly interfaces exist offers irrelevant protest. Finally, the utility of learning debugging can stand with an appeal to economy, religion of so many pragmatists and industrialists, without appeal to philosophy. Fortunately, Papert does not leave the argument at that and *does* extend his analysis with the rigor and elegance of philosophy. Papert had worked in Geneva with Piaget and in his book *Mindstorms: Children, Computers and Powerful Ideas* [2] generally admires Piaget's genetic epistemology– that children are exquisite and capable, even innately, epistemologists who busily bootstrap themselves by acquiring knowledge, acquiring strategies for acquiring knowledge and thinking about what knowledge is. Children, and by extension all of us adults too, are busily using our intuition to comprehend the world and, yes, our intuition is sometimes wrong. Intuition

needs debugging. While this does not yet directly address why intuition should be cut slack into our consideration, other than by its sheer inevitableness, the challenge is raised to build strong, correct intuitions.

Perhaps, the idea of debugging intuition hasn't strayed far from what the formalists seek while supposedly ignoring intuition and laboring on proofs. Without a serious sidetrack into why intuition is faulty, a few examples would suggest the overextending of patterns for blame in some circumstances. A model provides an abstraction workable in many scenarios, but not necessarily universally. Papert provides an example of a gyroscope [2]. The gyroscope is balanced on a narrow base and, intuitively, should fall over as objects with big bodies and tiny feet often do. While one may question the physics of the phenomena, one may also question why their often correct intuitions led them astray. To paraphrase Papert, one doesn't need proof that the gyroscope will stand; it's already obvious through experience that the gyroscope will remain upright while spinning. One needs resolution with their conflicted intuition. Working out and contemplating proofs may help provide an examination of intuition and means for its extension. The computer provides another platform for experimenting with intuition's expectations. Write a program that models something, perhaps the gyroscope or perhaps something simpler like the area of a right triangle. Run it. Are the results as expected? Finally, if all is not well, what changes might be made to the program to correct the behavior? Surprises could be the beacons of evolving intuition. The program acts as placeholder for memory as well as an instruction list to the machine on how the model should run. The neophyte programmer is not being asked to prove the program but to prove their intuition via the program. It's important to note that Papert's examples in *Mindstorms* needn't get dragged too far into the debate between formalism and software engineering. The initial models demonstrated with turtle graphics in the Logo programming language are small and geometric such as writing an instruction list that causes the turtle to sketch a triangle on the screen. The program already contains all the data it will ever need. It is a proof of itself, a set of one, and not rigorously a proof of the general case appliable across some larger set. However, the experience, the "ah-ha" of

the cause-and-effect experimentation feedback, betters an intuition that may subsequently be directed to many tasks, including proving the general case for a set.

It's not an obscure extension to stretch the computer as mathematics laboratory to a place to experiment with the phenomena mathematics is sometimes called on to model, like physics. Indeed the very idea of model suggests that the mathematics isn't just playing for its own sake. The word used in the Logo community is "microworlds" and it surely contains echos of virtual reality. The association of modeling and virtual reality is perhaps unavoidable since the latter must make strong use of the former. My analysis of virtual reality would be long, vitriolic and highly polemic. The counterpoint for that analysis would be art history and the suspicion that the development of virtual reality will parallel art history in its quest for mimesis (classical), its failure to attain it, an introspective period where it asks itself what it is instead of what it tries to be (modernism) and when it finally cloys of that will move happily forward as entertainment without its earlier pretensions. Perhaps the full thought will be forthcoming in another volume. But, I'll be more pleased if I can avoid ever having to write it. For now, modeling and virtual reality will be compared in light of computer literacy and say that two ideas imply different scopes of control for the user. Modeling is usually undertaken to help better understand a problem, like building a bridge or the epidemiology of an infectious disease, in an economical and safe manner. In ideal cases, the model is easily modified and scenarios compared.

Given the virtual reality insinuation of a "microworld," one should ponder which intuitions are being debugged? Could it be nothing more than the intuitions needed to write a correctly functioning program? Excluding sitting up, crawling, throwing things, walking, sledding, jumping and the like, my own early confrontations with my intuitions about gravity were not spurred by gyroscope; although, I recall finding them fascinating but acceptable in and of themselves. Of vivid recollection, because of the dilemma presented to my intuition, were the competing models of gravity I had from experiencing my own body, my early exposure to mathematical modeling of gravity and the virtual carton body of Wiley Coyote as it fell, along with enormous sandstone slabs, again and again down

unimaginably deep canyons. While any first hand experience I would have with the American desert southwest would wait two more decades, there were also some serious inconsistencies with rumors I had heard about feathers, cannonballs and leaning towers and my own experiments jumping off the tops of some sandy landslides along a nearby river bank. Why did Wiley and slab sometime hang for seconds in thin air before gravity kicked in? How come rates of acceleration of slabs, coyotes, roadrunners and the occasional acme explosive device suddenly change relative to each other? How was a slab able to act like a teeter-toter with the end bearing Wiley levered downward in midair without any seeable fulcrums? Why couldn't Wiley simply change direction by stepping off the slab, sideways, or jumping straight up prior to impact and thus save himself yet another flattening, which I clearly and correctly associated with rapid deceleration in the downward vector? The intuitive dilemmas presented exciting challenges despite the virtual source and, hopefully, most have been cleared up– for instance, with clarification via a .22 rifle and some beer cans during yet another childhood developmental stage, one not covered explicitly by Piaget. Aiming directly at the bottle fails to assassinate it, the bullet never truly travels a straight line. The slab and Wiley must hang in midair long enough for Roadrunner to pop his head in from the side of the screen. It must hang while Wiley and Roadrunner calmly look at each other, Roadrunner with an innocent insouciance and Wiley an indefatigable resignation at having not only missed his prey as surely as Evel Knievel missed the jump in Snake River Canyon, but with similar consequences. The slab must hang in mid air long enough for Wiley and Roadrunner to slowly turn, in unison, and share the sentiments drawn into their countenance with a stare out of the screen toward us the viewers. Finally, before the slab starts a decent of instantaneous acceleration, Roadrunner must "meep meep." The slab hangs because there is cartoon meaning in running amok of our intuition. The gravitational pause isn't license or convenience, but an intentional flaunt in the face of intuition meant to provoke the uncanny.

Models reflect. A model, like a popsicle house, abstracts attributes of an ideal model, a house, for consideration. Computers are math machines and as such are excellent for

directly exploring mathematical ideas, particularly those of combinatorics. Computers can also be used as musical instruments. Of course, one can tap at the keys, drum on its side or rip it from its cubicle foundation and send it noisily crashing through a plateglass window, sounding a resonating and shattering thud on the pavement ten stories below. The computer is full of oscillations, both symbolic and electrical, controllable by the program. Any computer can do this and most personal computers today come equipped with audio cards that will condition these oscillations to be played through speakers. A music synthesizer contains many "models" of analog sounds, but the musical performance stands on its own. The musician may quote or sample. The composition may even model an environment. A favorite free jazz trio of mine, The Fringe, does a mean swamp full of frogs using only analog instruments: a drum kit, saxophone, bass and the murky light of the Willow Jazz Club. The timbres are largely those expected of a jazz trio, but timings belong to a stagnant pool full of amphibians and a few flying pests. The musical composition doesn't require formal proof even though formal devices may be used in musical composition such as counterpoint or algorithmic music. The computer, program and all, is not acting as a proxy or reflector of an outside system via abstracting functions.

Without delving into the deep questions of what is judgment and what is aesthetics, more popular fodder for philosophical consumption, suffice to say that the correctness of some computer programs depends largely on aesthetics. While some heady algorithm based music like Bach's counterpoints or minimalist Terry Riley's "In C," the key and not the programming language, may be in search of a proof; the typical sweaty, dim, flesh filled club is likely to let it, and many other things, slide. Turtle graphics, the early graphics utilities of the Logo programming language and the core of Logo as a curriculum, appealed to the students' aesthetics. Programming a turtle to sketch on the screen presented definite formal problems such as, "how to describe a square in the Logo language." But combining squares with triangles, circles and other shapes to create the picture of a house, flower and setting sun makes an appeal to correctness somewhere other than the continuity of deductive steps for a set of axioms and lemmas. While such a goal and the freedom to explore an

openended question, or rather an openended answer, endears programming to those who'd prefer to be drawing over proving, Papert, following arguments of Poincaré, suggests that aesthetics plays an important role in mathematical insight.

> In reality, according to Poincaré, the mathematician frequently has to work with propositions which are false to various degrees but does not have to consider any that offend a personal sense of mathematical beauty [2].

So, let's review. Whether formalist or antiformalist, a description labeled as "intuition" usefully demarks a basic unit of acceptability. But where to ground this intuition? The sensuous. Sense.

And with the sensuous I'll make reluctant bedfellows of the logicians and the phenomenologists. Perhaps it's glitch philosophy to do so. I've been warned. But the defense will be that it's a consideration by which to view design decisions, particularly in designing the use of computers. The excuse will be that I'm a geek considering design and simply find it useful to do so. The link springs from phenomenology at the request of epistemology, particularly logic, but not necessarily logicians. With modern philosophy and its fulcrum on Kant, debate over thought's relation to truth and mechanisms for coming to have beliefs has been fruitful but mired in abstractions of thought-in-the-head. Rationalism and its operator deduction, empiricism and its operator induction or some combination of the two has failed to capture intuition, despite relying upon it. Empiricism may, mechanistically, come first and last with deduction providing economic thought crunching in between, perhaps even necessitating the creation of a subconscious since, as soon as deduction occurs at a conscious level, it morphs into empiricism, closing the metaphysical circle as we empirically consider our own deductive thoughts. The two explanations, deduction and empiricism, are analogous to homotropic functions; the space encompassed allows easy "you can get there from here" but the space remains locked up as abstractions, head thoughts and mental models. The dancer, the surfer, the saxophonist, the linebacker, the waiter deftly moving across a crowded floor with brunch for four balanced from his shoulder, for instance, exhibit refined proprioception. Psychology may create mental models explaining the spatial

sense of our own bodies and the coach may chalk-out a play in a blackboard schematic, but the body will make the moves. The phenomenologists, while kept company by artists and athletes, provide an eloquently written consideration of the embodied mind, a mind that does not exist without the body anymore than thought exists without intuition, intuition based on sense.

Edmund Husserl doesn't appear to be out to undermine science in *Ideas Pertaining to a Pure Phenomenology and to a Phenomenological Philosophy* [43] but rather aid science in it's failure to adequately handle intuition. Husserl takes science to task, smug in its increasingly self-referential successes, for it's lack of interest in "*de facto* sensuously intuitable shapes."

> It can then be seen, furthermore, that exact sciences and purely descriptive sciences do indeed combine but that they cannot take the place of the other, that no exact science, i.e. no science operating with ideal substructions, no matter how highly developed, can perform the original and legitimate tasks of pure description.

"Original" suggests a hierarchy and echos "first principles." The first principle, however, are not necessarily the root node of some tree on which the leaves depend, but "principles," the glue of the limbs making credible the link between each and every node in the tree, sensuous edges to the graphs of more traditional epistemology's mental models. While our belief of the proof may rest on intuition, should that be the interest at hand, it's discourse takes the form of "pure description."

> The most perfect geometry and the most perfect mastery of it cannot enable the descriptive natural scientist to express (in exact geometrical concepts) what he expresses in such a simple, understandable, and completely appropriate manner by the words "notches," "scalloped," "lens-shaped," umbelliform," and the like– all to them concepts which are *essentially, rather than accidentally, inexact* and *consequently* also non-mathematical.

It's a dangerous proposition to say, "but mathematics can't do this," since a new math is always inventable– just like a new sentence, particularly as math has become more comfortable with "close enough" estimations, which is at least since Newton and the Calculus. Maurice Merleau-Ponty, another eloquent phenomenological philosopher and key interpreter of Husserl, died in 1961. In 1959 at Citrően and in 1962 at Renault, mathematicians Paul de Casteljau and Pierre Bézier respectively developed the cubic curves known as Bézier curves for modeling curved automobile components. Today such cubic curves are standard in computer graphics. A surface of b-spline or Bézier curves would be the geometry of choice for modeling the shape of a scallop. Still, almost any child intuitively knows the complex shape of a scallop, but how many people can solve a third-order polynomial? Merleau-Ponty, who lived another twenty three years past Husserl, more generally states, "One must look for the sense of mathematical concepts in the life of consciousness on which they rest" [44].

In the case of electronic music, if not a much wider inclusion of genres, Nietzsche's challenge as to what has been comprehended by a scientific, formulaic estimation of music can be turned around. The formulas are the instruments of production; the "sense of mathematical concepts" finding a life in consciousness through the sensuousness of its music. Earlier, the paraphrase of Papert's analysis of Poincaré's view of the role of aesthetic in mathematics suggested another sensuousness, one found more directly in the formula themselves, without music. That will be left to the intuition of the individual; but, relating this thread back to computer literacy– the music provides a sensuous answer to the formula. Music, or some other appeal to aesthetic, can be explored with the powerful tool of formula with its predilections for deconstructing and identifying abstractions, patterns and connections. For those interested in the teaching of mathematics, the appeal to the motivating aesthetic provides the student a means, an end and a vehicle for exploring the intuitive connection between the two. Shift the challenge from finding the formula to producing a sound to understanding why a formula produces a sound. That the deductions correctly follow their axiomatic allowances may or may not be a goal, but understanding

why a deductive step was taken, ie. an appeal to intuition, should always be.

> Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection; we argue that insofar as it is successful, mathematics is a social, informal, intuitive, organic, human process, a community project [40].

Despite DeMillo *et al.* warmly describing mathematics with kind recognition of the role intuition plays, science and scientists, in general, were not asking phenomenology for its help. Positivism alone quenches many scientists' appetite for philosophical explanation. Even in Papert's essay "The Mathematical Unconscious," in which he strongly supports the role of intuition and aesthetics in mathematics, he writes, " The phenomenological view of abandonment is totally false," and goes on later in the paragraph, "This time the phenomenological view is even more misleading since the finished piece of work might appear in consciousness at the most surprising times, in apparent relation to quite fortuitous events" [2]. Papert is discussing the role of the subconscious in mathematical postulation, a sort of analytical computing area using some sort of memory swapping to communicate data with the consciousness– "abandonment" referring to the handing down by the consciousness to the subconscious mind's consideration. Our goal at present is not to reconcile entire intellectual movements and their progenitors, or even to enumerate their differences. Rather, if Papert wants his unconscious mind and his mental models, it shouldn't affect this argument; although, "subconscious mind" would better avoid confusion with the symptoms of high falls, syncope and Friday night drinking binges. Instead, allow the argument to continue its slow segue toward a punchline, or at least conclusion, that meaning is a process of construction.

Not only is the mind embodied but that body extends outward into our artifacts, and inward from them. Constructing meaning, constructing artifacts and constructing our bodies are one in the same. Artifacts can be inherited, along with their meanings. Computer literacy allows the user to construct meaning from and around a ubiquitous and potentially shackling technology. Within the Papert narrative of duality– reality and its model,

consciousness and its sidekick the unconscious mind- interesting claims are made about sense and mental models and its implication on the design of the Logo programming language. Papert's framework of duality, the design justification of the Logo programming language, and particularly turtle graphics, does touch on sense, particularly sense of the body. Borrowing terminology from Sigmund Freud and psychoanalysis, Papert argues that intimate and intuitive knowledge of one's own body is a well spring pattern for debugging mathematical problems through geometry– labeled "body syntonic" in *Mindstorms*. Programming becomes a sort of choreography of the turtle mappable to a choreography of the programmer's body. The program, for instance, could instruct the turtle pen to sketch a square on the screen. If the program fails to instruct the turtle to close the shape, perhaps by missing a ninety degree turn, the program can be literally stepped through across the ballroom floor, ie. debugging by dance.

This essay began by laboring over a debate played out by logicians. But its theme, if it's remained hidden, is on making art, really making art with and out of technology and the value for a literacy of technology. The lengthy discourse on intuition has intended to provide the bridge between "logic" and "art," if any bridge is really necessary; some may find no need for travel between the two words or, maybe, already find them significantly overlapped. Causality, at least in my naïve exposition, has been used synonymously with meaning, ie. causality is meaning. The cause of the tree falling in the forest without witness exists without meaning but to understand or speak of the cause of the tree's fall necessarily exercises meaning. Causality ultimately reduces to intuition and intuition to sense. Nothing of this immediately justifies excitement for computer literacy or art. The populist promotion of literacy is something of a cultural habit springing from enthusiasm for democracy and respect of the individual. Art, specifically the act of fabrication, similarly acts as an enabler of self-determination among a people overrun with consuming choices. The enthusiasm for computer literacy comes from a philosophy of athleticism, a philosophy amply demonstrated by the do-it-yourselfer.

But while appealing to a populist sense of equity, advocacy for computer literacy ap-

peals to individualism and a *laissez-faire* right to make one's self: that notion being found, in parts, not only in political rhetoric but in the constructivist education espoused by Seymour Papert or the bodily synthesis of phenomenologist Merleau-Ponty despite their differences on many points. Sense glues together intuition, causality and meaning with the body. Our philosophical investigation becomes enriched when the epistemological abstractions of mind are put into context of the body, ie. the embodied mind. Mind does not exist without the body. The mind of science fiction, a brain floating in solution, is a much different mind from that moving and sensing its way through the world. The edges between the world and the embodied mind are as transparent as they are crisp. Just as we might empirically consider our rational, deductive thoughts, the world doesn't stop at our skin; the inside edge of our bodies are inside the world as well. Furthermore, this prose becomes more powerful when considered in the context that our bodies and minds are dynamic and in a constant state of synthesis of what Merleau-Ponty calls "living meanings" in *Phenomenology of Perception.*

> Whether a system of motor or perceptual powers, our body is not an object of an 'I think', it is a grouping of lived-through meanings which moves toward its equilibrium. Sometimes a new cluster of meanings is formed; our former movements are integrated into a fresh motor entity, the first visual data into a fresh sensory entity, our natural powers suddenly come together in a richer meaning....

With such consideration, how might one consider artifacts as extensions of bodies? Outwardly, artifacts are extended body. When we write with pencil on paper, the pencil is a body extension communicating the forces at the tip to the hand just as the hand moves the graphite tip across the paper. Merleau-Ponty gives an example of the cane used by a blind man, an extending of tactile sensation.

> Hence my body can assume segments derived from the body of another, just as my substance passes into them; man is mirror for man. The mirror itself is the

> instrument of a universal magic that changes things into a spectacle, spectacles into things, myself into another and another into myself.

Timothy Leary noted that when he drove a Pontiac, he was a Pontiac. The surfer, the saxophonist, the gardener mowing the lawn: each like an insect with their antennas feeling the world around them, internalizing and likely modifying it. Knowledge at a distance, whether an ancient mariner retelling stories, even apocryphal ones, or the tele-epistemology of satellite communications. I sense, in some way, the thunderstorms in Hardwick, VT via my computer, the internet, a radar station in Burlington. The weather reporting mechanism extending my sense 4700km. Not only is the biological body, as intermediary of the senses, a component of the mind, so too is artifact as an extension of the body:

Sometimes pretty language is useful. One could protest that while I might describe viewing of artifact as some sort of transcendental, hippy-dippy fusing of consciousnesses, a lot is left up in the air after the gaze. Say I hike out to view some five thousand year old petroglyphs. For whatever mind-meld transpires, I really have no clue about the important questions that artist was considering. I can only conjecture the motivations and meanings. True. But perhaps I am expecting too much; like the freshman's acid trip, they expected the gates of heaven but only got a dizzy headache and an upset stomach.[3] But something was learned in the gaze. And something is learned in the feel of the rock, a sense I now share with the long departed progenitor of that creative spark.

Knowledge is not truth; knowledge is belief of what might be true. The old aphorism, "I know what I know," suggests that we act on the best possible information. It suggests a skepticism in what we know and its relation to truth. And, it suggests that it's easier to know the present, to know what we have constructed into ourselves than to actually know the past. At the same time that I can study artifacts of the past, I can only understand them in the present. The past may strongly affect my present, as in a fascist sentimentality. I may

[3]Or consider the countless jokes over abstract, particularly minimalist, paintings. But the joke's on the sarcastic skeptic who fails to recognize that the reified abstraction is realistic while the mimesis of photorealism is the fake of reality, except, again, that they are really fakes and really paintings.

quote or cite, but the work carries on into the present. What I understand best is the lesson I have constructed into myself; my conjectures live in the present.

Artifacts impart a hegemony of the body as well. Prisons or handcuffs are obvious examples. Door knobs, the height of a chair or the length of a symphony are others. Designers talk about the constraints of designs. Those constraints are often factored into a design for good reasons such as safety. An elevator door has no handles and I am hard pressed to open it while the carriage is in motion. Computers and computer software are now full of constraints. Some of this arises from the important engineering tasks relying on software. Should my program encounter data it's unequipped to handle, better that a fallback mechanism be provided that allows a graceful failure while the proverbial airplane and nuclear powerplant may go merrily on their way. Constraints are put into software to protect us from frustration. The concern is that a constraint on frustrating literacies is equivalent to illiteracy.

Chapter 6

Conclusion (Open ended of course)

This thesis makes an argument for computer literacy as a generally and widely useful skill, ie. a basic competency in contemporary education. Although the line is not sharp, "user friendly" is presented as often at odds with "computer literacy" by embracing design strategies that obfuscate the underlying operation of the computer and creating obstacles to users taking full control of the computer's potential. The advocates of user friendly designs are often software companies seeking to increase market share, companies that are all too willing to be stingy with their knowledge at the expense of a more dependent user. While corporate hegemony motivates my interest in computer literacy, the arguments are built on top of those by advocates for constructionist education. Computer programming, as opposed to multimedia drill-and-quiz style software, lends itself to hands-on education. Computer programming provides an ideal microcosm for learning modeling and modeling can be found at the heart of problem solving. Computer programming inevitably involves debugging and debugging is a problem solving skill that transcends a utility limited to computer science or computers.

When confronted with learning a new skill or technology, one is likely to start the exploration somewhere in the middle of things; the point here is to design into the machine the ability to move up and drill down on the mechanism to a level of understanding that fits our motivations. The design of Smalltalk with an IDE that reveals the very code running its virtual machine is an excellent example of the principle I'm advocating. Open source platforms such as Linux are also an excellent example of this principle, but Linux, albeit

open source, remains opaque, a hurdle with a significant learning curve. Logo and LogoRhythms provide some alternative teaching languages that hope to help students in the direction of having the skills to negotiate more confidently the mess that often accompanies free software.

What LogoRhythms is: a simple audio API in the Logo language that allows one to explore programming concepts in the audio domain. The emphasis is on basic manipulation of digital sound wave data starting with sinusoids. LogoRhythms is open source, free and cross platform. LogoRhythms follows in the model of constructionist philosophy of education where students are viewed as epistemologists coming to grip with what their knowledge is and taking responsibility for its construction through hands-on experience and empirical experimentation. Open source also allows the student to deconstruct; it provides the mechanical translucency that enables the curious to push their envelope of causal understanding.

What LogoRhythms is not: a full featured music synthesis API looking to compete with the likes of Chuck or Max/MSP [45][5]. Beyond lacking functionality such as a scheduler, it has several serious shortcomings: lack of cross platform threading, a bulky notion of arrays that are really linked lists and lack of a mixer. Of course, given the potentials of glitch, there's no reason to completely rule LogoRhythms out as a vehicle for generating noteworthy music. But, its praxis is firstly educational.

What's missing here is a good field study, ie. I had hoped to produce a narrative providing demonstration of the API being used by its intended audience. Such demonstration provides reflection that the project can accomplish what it purports as well as highlighting strengths, weaknesses and potential improvements. More importantly though, the project was meant to serve a real need, to help slow the erosion of computer literacy and give kids (or adults) the opportunity to see that programming isn't so arcane and difficult a skill as the overuse of the term "guru" would suggest. Given the bureaucratic difficulty of organizing even the simplest of user studies within the school system, the project really needs

to be a collaboration with an enthusiastic educator already on the inside of the system.[1] The project is open source– more than reasonable given the gracious source of funding from a Canadian public university. But as Eric S. Raymond points out in his oft cited "The Cathedral and the Bazaar," open source projects fail or succeed on the enthusiasm of its user base, enthusiasm coupled with strong leadership [46].

Now then, in what direction might this inquiry continue?

This thesis has provided a three pronged approach toward an examination of computer literacy. First, there is the engineering approach via the creation of LogoRhythms, what I've also called experimental archeology in that LogoRhythms's base language of Logo is a product of the heyday of computer literacy brushed up to take advantage of contemporary hardware capabilities. Second is a historical approach both tracing the evolution of the emphasis in HCI literature from "computer literacy" to "user friendly" as well as examining who made the early radical revolutions in computer science and where their backgrounds lay. Third is a philosophical argument that suggests a link between intuition and causality, the body and knowledge, artifacts and hegemony and mechanical transparency as a democratic design imperative. Underlying much of these arguments is an assumption that many people in fact do not understand how their computer's operate or have faulty explanations for the computer's mechanics, that the users may feel a level of alienation from a technology that they're fully aware has been dumbed down to compensate for their simple understandings. Where philosophy may persuade by a convincing argument, anthropology can reinforce with empirical study. How do people fetishize the technology? What are the folk explanations given by the machine's users for its mechanical operation? For instance, the brief user study described in chapter 4 included a question, "what numbers can a computer not represent?" Among the myriad of responses given such as imaginary numbers, rational numbers, really big numbers, Roman numerals– each arguably incorrect responses- none of the first year computer science or computer music students noted that the computer works with discrete numbers, it's numberlines filled with an infinitum of sizable gaps.

[1] To get access to a classroom in Hawai'i, for instance, involves submitting one's fingerprints to the FBI.

HCI literature is often mired in the challenge test as evaluation method: how fast and how efficiently can a user do something. Focus groups, in my experience frequently used by corporate interface designerd, do attempt to query end users on how they actually used, felt and reacted to a technology. But how do the users use *in situ*? Ethnographies of computer users do exist. Some dwell in heady debate of sociological theory such as Jeffery Alexander's "The Sacred and Profane Information Machine" that examines, via public discourse, beliefs about the computer's potential as salvation or apocalyptic damnation [47]. Field ethnographies have also been conducted among computer professionals. Examples include Gary Lee Downey's *The Machine in Me: An Anthropologist Sits among Computer Engineers* and Georgina Born's *Rationalizing Culture: IRCAM, Boulez, and the Institutionalization of the Musical Avant-guarde* [48][49]. But these volumes deal largely in questions of culture and power and the deciphering of semiotics, less on the simple documentation of beliefs and practices, enumerations less encumbered by debate over emic versus etic perspective. Other studies do employ ethnographic surveys to computer users in an effort to describe naturalistically how the users actually debug [50] or make software engineering design decisions [51]. I say "naturalistically" as opposed to "prescriptively," ie. how people do the job versus what they are told is an ideal way to do the job. Better examples of such studies exist in mathematics, perhaps not surprising since mathematics is taught to most students at all levels of their education with concomitant concern that the teaching be effective. Capon and Kuhl studied grocery store mathematics inside (and outside) the grocery store [52], Carraher *et al* looked at street mathematics of primary school dropouts working on the streets of Brazil [53] and Masingila examined algorithms used by construction workers laying carpeting for a floor covering company in the western United States [54]. These papers offer excellent examples of ecologically valid data collection efforts. While they focus on algorithms used day-to-day, the study I propose here would focus on the mechanical operation, including programming, of the computer, not unlike the interview based studies of Piaget for the bicycle [15]. How do just-plain-folks believe their computer operates?

My own view motivating the design of LogoRhythms parallels the model set by the tiny violins used by preschoolers studying classical music via the Suzuki method, a smaller size appropriate to the bodies of smaller players but fully functional as instruments. Most programming languages have a specific function for which they are adapted: Lisp to AI research, Java to enterprise level commercial applications, PHP for webpages, Logo to teaching, etc... Dane Bjarne Stroustrup writes, "If you think C++ is difficult, try English" [55]. Suzuki posits that since all children can learn their natural language, a harder task than violin, all children can learn the violin... and learn it well [56]. Turning Stroustrup's quip around– why not teach children C++? Or maybe just C? Perhaps instead of creating LogoRhythms, I might have written a compendium of "101 Rainy Day Activities for Your Kids with C++." Without delving into the language's many complexities, it doesn't seem unreasonable to introduce variables, arithmetic, sequential execution of statements, conditionals and loops, the capabilities necessary for a language to be Turing complete [57]. While it might be harder to explore the digital audio themes of LogoRhythms in C++, no doubt creative activities can be imagined that help provide a good imperative language foundation, for instance, ascii art drawings that show recursive patterns. In a nutshell, from a didactic point of view, what's so bad about a dangling pointer and a crash? The deeper meaning and value of computer science isn't necessarily to make functioning applications, but rather to explore, sensorially, a formal logic at high speeds. And, to echo Seymour Papert, to teach the widely useful skill of debugging. To echo Alan Kay, to provide an environment in which to explore problems by modeling them, to find solutions through rapid, easily tweaked digital experimentation (which to echo Richard Feynman should never come to completely replace good old fashion empiricism with real material, *prima facia* [58].)

But the most important question, of course, is how does LogoRhythms perform with real kids and other neophyte computer users? Logo has long proven itself and continues to exist commercially in LCSI's Microworlds and in open source format via UCB Logo among others. Whether LogoRhythms is useful to its target audience remains to be seen and is a question now best put to the field.

Bibliography

[1] J. C. S. Bauer, *Das erste Zweirad fuhr in Mannheim - Die fruheste Beschreibung der Drais'chen laufmachine 1817*. unknown, 1819.

[2] S. Papert, *Mindstorms: Children Computer and Powerful Ideas*, 1st ed. Basic Books, Inc., 1980.

[3] Microworlds. (2005, October). [Online]. Available: http://www.microworlds.com/solutions/index.html

[4] M. Puckette, "Max at seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, Winter 2002.

[5] D. Zicarelli, "How i learned to love a program the does nothing," *Computer Music Journal*, vol. 26, no. 4, pp. 41–51, Winter 2002.

[6] B. Harvey, *Computer Science Logo Style, Vol 3: Advanced Topics*, 1st ed. The MIT Press, 1987.

[7] M. Guzdial. (2006, March) Teaching programming with music: An approch to teaching young students about logo. [Online]. Available: http://el.media.mit.edu/Logo-foundation/pubs/titleindex.html

[8] "Big isle schools on list," *Hawaii Tribune Herald*, September 2005.

[9] B. Harvey. (2006, March) Ucb logo download page. [Online]. Available: http://www.cs.berkeley.edu/~bh/

[10] (2006, March) Portaudio home page. [Online]. Available: http://www.portaudio.com/

[11] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterlink, *Numerical recipes in C : the art of scientific computing*, 2nd ed. Cambridge University Press, 1992.

[12] R. B. Dannenberg, "Machine tongues xix: Nyquist, a language for composition and sound synthesis," *Computer Music Journal*, vol. 21, no. 2, pp. 50–60, Fall 1997.

[13] *Nyquist Reference Manual, Version 2.22*, CMU, 2002, by R. B. Dannenberg.

[14] D. A. Norman, *The Design of Everyday Things*, 1st ed. Basic Paperback, 2003, originally published 1988.

[15] J. Piaget, *The Child's Conception of Physical Causality*. Harcourt Brace and Company, 1930.

[16] K. Cascone, "The aesthetics of failure: Post-digital tendencies in contemporary computer music," *Computer Music Journal*, vol. 24, no. 4, pp. 12–18, Winter 2000.

[17] C. Stuart, "Damaged sound: Glitching and skipping compact discs in the audio of yasunao tone, nicholas collins and oval," *Leonardo Music Journal*, vol. 13, pp. 47–52, 2003.

[18] B. Pfaffenberger, "The social meaning of the personal computer: Or, why the personal computer revolution was no revolution," *Anthropology Quarterly*, vol. 61, no. 1, pp. 39–47, January 1988.

[19] ACM. (2006, April) Chi. [Online]. Available: http://portal.acm.org

[20] M. Guzdial and E. Soloway, "Computer science is more important the calculus: The challenge of living up to our potential," *SIGCSE Bulletin*, vol. 35, no. 2, pp. 5–8, June 2003, invited Editorial.

[21] W. Bush, V. Donahue, and K. Kelly, "Pattern recognition and display characteristics," *IRE Transactions on Human Factors in Electronics*, vol. 1, pp. 11–20, March 1960.

[22] B. F. Green Jr, "Computer languages for symbolic manipulation," *IRE Transactions on Human Factors in Electronics*, vol. 2, pp. 3–7, March 1961.

[23] H. Freiberger and E. Murphy, "Reading machines for the blind," *IRE Transactions on Human Factors in Electronics*, vol. 2, pp. 8–19, March 1961.

[24] T. Marill, "Automatic recognition of speech," *IRE Transactions on Human Factors in Electronics*, vol. 2, pp. 8–19, March 1961.

[25] M. Minsky, "A selected descriptor-indexed bibliography to literature on aritifical intelligence," *IRE Transactions on Human Factors in Electronics*, vol. 2, pp. 39–55, March 1961.

[26] R. L. Deininger, "Desirable push button characteristics," *IRE Transactions on Human Factors in Electronics*, vol. 1, pp. 24–29, March 1960.

[27] F. A. Brooks, "Operational sequence diagrams," *IRE Transactions on Human Factors in Electronics*, vol. 1, pp. 33–24, March 1960.

[28] ——, *The Mythical Man Month*, anniversary ed. ed. Addison-Wesley Pub. Co., 1995.

[29] I. E. Sutherland, "Sketchpad: A man-machine graphical communication system," in *AFIPS Conference Proceedings, Volume 23, of the Spring Joint Computer Conference*, 1963.

[30] W. K. English, D. C. Engelbart, and M. L. Berman, "Display-selection techniques for text manipulation," *IEEE Transactions on Human Factors in Electronics*, vol. 8, no. 1, pp. 5–15, March 1967.

[31] R. W. Scheifler and J. Gettys, "The x window system," *ACM Transactions on Graphics*, vol. 5, no. 1, pp. 5–15, March 1967.

[32] J. Callahan, D. Hopkins, M. Weiser, and B. Schneiderman, "An empirical comparison of pie vs. linear menus," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1988.

[33] C. H. Museum. (2006, March) Computer history museum - timeline. [Online]. Available: http://www.computerhistory.org/timeline/

[34] E. Dijkstra, "On the cruelty of really teaching computer science," *Communications of the ACM*, vol. 32, no. 12, pp. 1398–1404, December 1989.

[35] A. C. Kay, "The early history of smalltalk," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 23–95, March 1993.

[36] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

[37] W. A. Schloss and D. A. Jaffe, "Intelligent musical instruments: The future of musical performance or the demise of the performer?" *INTERFACE Journal for New Music Research*, vol. 22, no. 4, pp. 183–193, 1993.

[38] M. or Astroman, "Experiment zero," compact disc, January 1996.

[39] F. Nietzsche and W. Kaufman (trans.), *The Gay Science*. Vintage Books, 1974 (1887).

[40] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, no. 5, pp. 271–280, May 1979.

[41] E. Dickinson (1830-86), *The Complete Poems of Emily Dickinson*. Back Bay Books UK, 1976.

[42] H. Poincaré, *The Value of Science*, 1st ed. Dover Publications, 1958.

[43] E. Husserl and F. Kersten (trans), *Ideas Pertaining to a Pure Phenomenology and to a Phenomenological Philosophy*. Martinus Nijhoff Publishers, 1982 (1913).

[44] M. Merleau-Ponty and J. M. Eide (trans), *The Primacy of Perception*. Northwest University Press, 1964.

[45] G. Wang and P. Cook, "Chuck: A concurrent, on-the-fly, audio programming language," in *Proceedings of the 2003 International Computer Music Conference*, 2003.

[46] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 1999.

[47] J. Alexander, "The sacred and profance information machine: Discourse about the

computer as ideology," *Archives de sciences sociales des religions*, vol. 69, pp. 161–170, janvier-mars 1990.

[48] G. L. Downey, *The Machine in Me: An Anthropologist Sits among Computer Engineers*. Routeledge Press, 1998.

[49] G. Born, *Rationalizing Culture: IRCAM, Boulez, and the Institutionalization of the Musical Avant-guarde*. University of California Press, 1995.

[50] L. Gugerty and G. M. Olson, "Debugging by skilled and novice programmers," in *ACM SIGCHI Proceedings*, 1986.

[51] D. C. Rine and R. M. Sonnemann, "Investments in reusable software. a study of software reuse investment success factors," *Journal of Systems and Software*, vol. 41, pp. 17–32, 1998.

[52] N. Capon and D. Kuhn, "Logical reasoning in the supermarket: Adult females' use of a proportional reasoning strategy in an everyday context," *Developmental Psychology*, vol. 15, no. 4, pp. 450–452, 1979.

[53] J. Lave, *Cognition in Practice: Mind, mathematics and culture in everyday life*. Cambridge University Press, 1988.

[54] J. O. Masingila, "Mathematics practice in carpet laying," *Anthropology and Education Quarterly*, vol. 25, no. 4, pp. 430–462, 1994.

[55] B. Stroustrup, *C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.

[56] S. Suzuki and W. t. Suzuki, *Nurtured by Love: A New Approach to Education*. Exposition Press, 1969.

[57] K. C. Louden, *Programming Languages: Principles and Practice*. Brooks/Cole–Thomas Learning, 2003.

[58] R. P. Feynman, "An outsider's inside view of the challenger inquiry," *Physics Today*, vol. 41, pp. 26–36, Febuary 1988.

Appendix A

AlphabetSynth Source Code

Below is the Logo source code for the AlphabetSynth application discussed in Chapter 2.

A.1

```
;; qsort
to sort :data

  local [lft rght pivot next]

  if equal? 1 count data [ output data ]
  if equal? 0 count data [ output data ]

  make "lft []
  make "rght []
  make "pivot first (first data)
  make "lft fput (first data) lft
  make "data butfirst data

  repeat count data [
    make "next first first data
    if less? next pivot [ make "rght fput (first data) rght ]
    if equal? next pivot [ make "rght fput (first data) rght ]
    if less? pivot next [ make "lft fput (first data) lft ]
    make "data butfirst data
  ]

  make "lft sort lft
  make "rght sort rght

  output cons rght lft

end
```

```
;; believe it or not, this doesn't seem to be part of UCB Logo
to cons :list1 :list2

  if empty? list2 [ output list1 ]

  make "list1 lput first list2 list1
  make "list2 butfirst list2
  make "list1 cons list1 list2

  output list1

end


;; from Harvey
to tree :key :children
  local [ node ]
  output fput key children
end


;; from Harvey
to leaf :datum
  output tree data []
end


;; from Harvey
to btree :data
  if empty? data [ output [] ]
  if empty? bf data [ output data ]
  output btreehelper (int (count data)/2) data []
end


;; from Harvey
to btreehelper :c :in :out
  if equal? c 0 [
    output tree (first in) (list (btree reverse out) (btree bf in))
  ]
  output btreehelper (c-1) (bf in) (fput first in out)
end


to synth

  local [ w1 w2 w3 freq ]
```

```
  ;; add a combintion of sinusiods
  make "freq 440
  make "amp 1
  make "w1 sinewave freq
  make "i 2
  repeat 2 [
    make "w2 sinewave freq * i
    make "i i*2
    make "amp amp/1.4
    make "w2 volume w2 amp
    make "w1 combinewaves list w1 w2
  ]
  make "w2 sinewave freq/2
  make "w2 volume w2 amp/2
  make "w1 combinewaves list w1 w2

  ;; add a combintion of trianglewaves
  make "freq 430
  make "amp .6
  make "w2 trianglewave freq
  make "w2 volume w2 amp
  make "w1 combinewaves list w1 w2
  make "i 2
  repeat 2 [
    make "w2 trianglewave freq * i
    make "i i*2
    make "amp amp/1.4
    make "w2 volume w2 amp
    make "w1 combinewaves list w1 w2
  ]
  make "w2 trianglewave freq/2
  make "w2 volume w2 amp/2
  make "w1 combinewaves list w1 w2

  make "n noise 1
  make "n volume n .05
  make "w1 combinewaves list w1 n

  make "w1 normalizewave w1
  make "w1 evenwt w1

  output w1
end


to organ :base :fade
  make "wave synth
  output organhelper (base * (1 / ln 2)) :fade :wave [] 0
end
```

```
to organhelper :base :fade :wave :data :i
   local [ env note freq ]

   make "env list [.9 25] lput :fade [0]
   make "freq (base * (power 2 i) * (ln 2))
   if less? 2048 freq [ output data ]

   make "env list freq env
   make "note lput env [soundwt :wave]
   make "note list freq note

   make "data fput note data
   make "i i + 1/12

   output organhelper base fade wave data i

end


to organkeys :base
  make "base first :list.synth
  make "base base * (1 / ln 2)
  make "c ascii readchar
  output (base * (power 2 (:c - 97)/12) * (ln 2))
end

to startsynth :func.synth :list.synth :func.getkey :list.getkey
  local [ freq lasttime deltatime loop cmd b ]

  make "cmd []

  make "b btree sort (apply :func.synth :list.synth)

  ;;  looping has three states 0:off and clear, 1:recording, 2:looping
  make "looping 0

  forever [
   make "key apply :func.getkey :list.getkey

    test equal? c 58
    iffalse [ make "cmd last lookup :key b ]
    iftrue [
      if equal? looping 2 [
        make "looping 0
      ]
      if equal? looping 1 [
make "looping 2
      ]
      if equal? looping 0 [
```

```
        make "lasttime time
   make "delta 0
make "loop []
        make "looping 1
      ]
    ]

    if equal? looping 1 [
        make "loop fput cmd :loop

        make "delta (time - lasttime)
        make "lasttime time
        make "r [rest]
        make "r lput :delta r
make "loop fput :r :loop
    ]
    ifelse equal? looping 2 [
test empty? loop
        iffalse [ forever [ playloop loop :wave ] ]
        make "looping 0
    ] [
      test empty? cmd
      iffalse [
        test equal? c 58
  iffalse [ run cmd ]
      ]
    ]

  ]

end


to playloop :data :wave
  test empty? :data
  iffalse [
    run last :data
    playloop butlast :data :wave
  ]
end


to lookup :code :tree
  output lookuphelper :code :tree []
end


to lookuphelper :code :btree :closest
  local [ next less more ]
```

```
  make "this getNodeKey :btree

  if empty? :closest [ make "closest getNode :btree ]
  if equal? :code :this [ output getNode :btree ]
  if isleaf? :btree [ output closest ]

  ifelse less? :code :this [
    test empty? getlessbranch :btree
    iffalse [
      make "less getNodeKey getLessBranch :btree
      if updateclosest? :code :less first closest [
        make "closest getNode getLessBranch :btree
      ]
      test isleaf? getLessBranch :btree
      iffalse [ make "closest lookuphelper :code getLessBranch :btree :closest ]
    ]
  ] [
    test empty? getmorebranch :btree
    iffalse [
      make "more getNodeKey getMoreBranch :btree
      if updateclosest? :code :more first closest [
        make "closest getNode getMoreBranch :btree
      ]
      test isleaf? getMoreBranch :btree
      iffalse [ make "closest lookuphelper :code getMoreBranch :btree :closest ]
    ]
  ]

  output closest
end


to updateclosest? :code :next :current
  output less? abs (code - next) abs (code - current)
end


to isleaf? :x
  output less? count :x 2
end


to abs :x
  ifelse less? x 0 [ output minus x ] [ output x ]
end


to getNode :btree
  output first :btree
end
```

```
to getNodeKey :btree
  output first first :btree
end


to getNodeValue :btree
  output last first :btree
end


to getLessBranch :btree
  output first butfirst :btree
end


to getMoreBranch :btree
  output last :btree
end


;; start the organ alphasynth
load "synthdb.lg
startsynth [organ ?1 ?2] [220 128] [organkeys ?1] [220]
```