The ChucK Audio Programming Language "A Strongly-timed and On-the-fly Environ/mentality"

Ge Wang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Perry R. Cook

September 2008

Abstract

The computer has long been considered an extremely attractive tool for creating, manipulating, and analyzing sound. Its precision, possibilities for new timbres, and potential for fantastical automation make it a compelling platform for expression and experimentation - but only to the extent that we are able to express to the computer what to do, and how to do it. To this end, the programming language has perhaps served as the most general, and yet most precise and intimate interface between humans and computers. Furthermore, “domain-specific” languages can bring additional expressiveness, conciseness, and perhaps even different ways of thinking to their users.

This thesis argues for the philosophy, design, and development of ChucK, a general-purpose programming language tailored for computer music. The goal is to create a language that is *expressive* and *easy to write and read* with respect to time and parallelism, and to provide a platform for precise audio synthesis/analysis and rapid experimentation in computer music. In particular, ChucK provides a syntax for representing information flow, a new time-based concurrent programming model that allows programmers to flexibly and precisely control the flow of time in code (we call this “strongly-timed”), and facilities to develop programs *on-the-fly* - as they run. A ChucKian approach to *live coding* as a new musical performance paradigm is also described. In turn, this motivates the Audicle, a specialized graphical environment designed to facilitate on-the-fly programming, to visualize and monitor ChucK programs in real-time, and to provide a platform for building highly customizable user interfaces.

In addition to presenting the ChucK programming language, a history of music and programming is provided (Chapter 2), and the various aspects of the ChucK language are evaluated in the context of computer music research, performance, and pedagogy (Chapter 6). As part of an extensive case study, the thesis discusses ChucK as a primary teaching and development tool in the Princeton Laptop Orchestra (PLOrk), which continues to be a powerful platform for deploying ChucK 1) to teach topics ranging from programming to sound synthesis to music composition, and 2) for crafting new instruments, compositions, and performances for computer-mediated ensembles. Additional applications are also described, including classrooms, live coding arenas, compositions and performances, user studies, and integrations of ChucK into other software systems.

The contributions of this work include the following. 1) A time-based programming mechanism (both language and underlying implementation) for ultra-precise audio synthesis, naturally extensible to real-time audio analysis. 2) A non-preemptive, time/event-based concurrent programming model that provides fundamental flexibility and readability without incurring many of the difficulties of programming concurrency. 3) A ChucKian approach to writing code and designing audio programs on-the-fly. This rapid prototyping mentality has potentially wide ramifications in the way we think about coding audio, in designing/testing software (particular for real-time audio), as well as new paradigms and practices in computer-mediated live performance. 4) The Audicle as a new type of audio programming environment that combines live development with visualizations. 5) Extended case studies of using, teaching, composing, and performing with ChucK, most prominently in the Laptop Orchestra. These show the power of teaching programming via music, and vice versa - and how these two disciplines can reinforce each other.

Acknowledgments

In developing the works described in this dissertation (as well as the document itself), I am indebted to a great many people for teaching, help, guidance, and encouragement throughout. I am deeply grateful to Perry Cook for his teaching, mentorship, friendship, and for granting me the freedom to explore new (sometimes crazy-seeming) directions while providing the guidance and encouragement to help me to see things through. Immense gratitude to Dan Trueman for his always considerate and generous help and guidance, and (to both Dan and Perry) for trust and confidence in me to help develop the laptop orchestra. I have been incredibly fortunate to work with Paul Lansky, Ken Steiglitz, Brian Kernighan, Bede Liu, Wayne Wolf, Kai Li, and Andrew Appel. I thank them for guidance from and to many directions. Very special thanks to Roger Dannenberg, who has provided invaluable guidance and support throughout my graduate career. Deep thanks to Melissa Lawson for taking care of us graduate students since before we arrived.

Many aspects of this work have benefited immensely from people working hard together. It continues to be my great honor to work with folks at the SoundLab: Ajay Kapur, Ari Lazier, Philip Davidson, Spencer Salazar, Tae Hong Park, Adam Tindale, Ahmed Abdallah, Paul Botelho, Matt Hoffman, Jeff Bernstein. I am truly grateful to Ananya Misra – together weve pulled through many projects, demos, papers, and presentations. I am incredibly honored to work with Scott Smallwood, whether it is surviving PLOrk or composing/performing together. Heartiest thanks to comrade Rebecca Fiebrink, for being amazing in everything we do.

Many thanks to Adam Finkelstein, Szymon Rusinkiewicz, Tom Funkerhouser, and the always illuminating folks in the Graphics Lab. To Ed Felten, Alex Halderman, Frances Perry, and Shirley Gaw, Limin Jia, Yong Wang, and Jay Ligatti for security and support. Special thanks to CS Staff for amazing support, and to all faculty, staff, and students at the Princeton Computer Science Department.

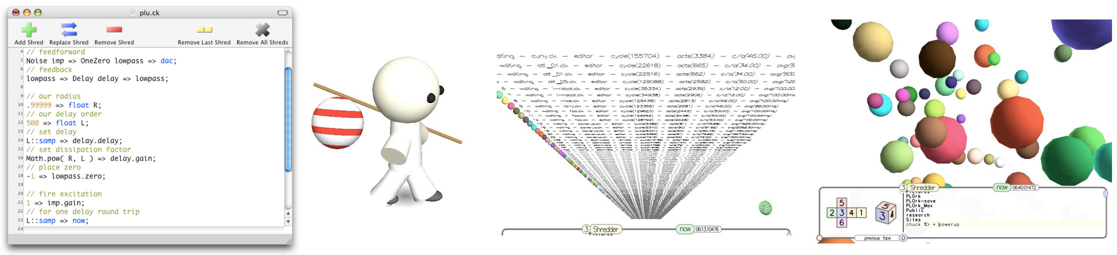
I have also been fortunate to collaborate with many great mentors and colleagues outside of Princeton. My deep thanks to Jon Appleton for his support and friendship. To Larry Polansky and Kui Dong for their encouragement, and for inviting me to teach the graduate seminar at Dartmouth in 2007. To the graduate students in the Electro-acoustic Music program; to Yuri Spitsyn for his friendship and guidance. Special thanks to Gary Scavone for being a great colleague (even though we rarely see each other in person), George Tzanetakis for endlessly fascinating discussions on software design for audio systems, Georg Essl for being a wonderful colleague and friend, Nic Collins and Shawn Decker for encouragement and interest in this work, Brad Garton for his wonderful ideas and support. My profound thanks to Max Mathews and John Chowning for continued inspiration and encouragement, as well as to Chris Chafe, Julius Smith, Jonathan Berger, Chryssie Nanou, Rob Hamilton, Jonathan Abel, Bret Ascarelli, Jieun Oh, and the amazing 2007-2008 MA/MST students, as well as everyone else at CCRMA and the Stanford Music Department for their deep support and belief in me. I profusely thank the ChucK users and developers communities, especially Kassen and Kijjaz, as well as Nick Collins, Alex McLean, and other fellow live coders and colleagues at TOPLAP. I leave out many wonderful folks to whom I am indebted and whom I keep at heart.

This research was supported by funding from the National Science Foundation (0101247, 0509447, 9984087) and by Interval Research and the Arial Foundation.

Finally, thanks to my grandparents who are responsible for the good in me, my parents for standing behind me and encouraging my interests, to Manman for her sacrifices and undying support.

Contents

List of Figures



```
// feedforward
Noise imp => OneZero lowpass => dac;
// feedback
lowpass => Delay delay => lowpass;

// our radius
.99999 => float R;
// our delay order
500 => float L;
// set delay
L::samp => delay.delay;
// set dissipation factor
Math.pow( R, L ) => delay.gain;
// place zero
-1 => lowpass.zero;

// fire excitation
1 => imp.gain;
// for one delay round trip
L::samp => now;
```

Figure 1.1: Some ChucKian things.

Chapter 1

Introduction and Motivation

"The old computing is about what computers can do. The new computing is about what *people* can do." - Ben Shneiderman

1.1 Problem Statement

The computer has long been considered an extremely attractive tool for creating and manipulating sound [54, 93]. Its precision, possibilities for new timbres, and potential for fantastical automation make it a compelling platform for experimenting with and making music - but only to the extent that we can actually tell a computer what to do, and how to do it.

A program is a sequence of instructions for a computer. A programming language is a collection of syntactic and semantic rules for specifying these instructions, and eventually for providing the translation from human-written programs to the corresponding instructions computers carry out. In the history of computing, many interfaces have been designed to instruct computers, but none have been as fundamental (or perhaps as enduring) as programming languages. Unlike most other classes of human-computer interfaces, programming languages don't directly perform any specific "end-use" task (such as word processing or video editing), but instead allow us to build software that might perform almost any custom function. The programming language acts as a mediator between human intention and the corresponding bits and instructions that make sense to a computer. It is the most general and yet the most intimate and precise tool for instructing computers.

Programs exist on many levels, ranging from assembler code (extremely low level) to high-level scripting languages that often embody more human-readable structures, such as those resembling spoken languages or graphical representation of familiar objects. *Domain-specific languages* retain general programmability while providing additional abstractions tailored to the domain (e.g., sound synthesis).

Yet, even within the domain of audio programming, there is a staggeringly vast range of tasks that one may wish to perform (or investigate), ranging from methods for sound synthesis, physical modeling of real-time world artifacts and spaces (e.g., musical instruments, environmental sounds), analysis and information retrieval of sound and music, to mapping and crafting of new controllers and interfaces (both software and physical) for music, algorithmic/generative processes for automated or semi-automatic composition and accompaniment, real-time music performance, to many others. Moreover, within each of these areas, there lies unbounded variation in programming approaches, styles, and demands on the tools (e.g., ability to create/run real-time programs).

Furthermore, audio programming, in both computational acoustics research and in music composition and performance, is necessarily an experimental and empirical process; it requires rapid experimentation, verification/rejection/workshoping of ideas and approaches, and takes the forms of both short-term and sustained prototyping. It can greatly benefit from the ability to modify, or even create, parts of the software system "on-the-fly" – as it runs. We believe that rapid prototyping, in and of itself, is a uniquely useful approach to programming audio, with its own benefits (and different ways of thinking about problems).

Faced with such a wide gamut of possibilities and demands, how do we go about thinking about and designing a general programming tool to address these aspects of expressive programmability, rapid prototyping, readability? This is the problem statement, and this dissertation addresses its various facets in terms of a new programming language, called ChucK, and chronicles its design, ideas, and applications.

In addition to our desire to address the problems stated above, we are also motivated in providing new tools for computer science, computer music pedagogy, and for exploring with new musical paradigms. We believe an audio-centric language such as ChucK should be useful to both novices learning about the domain, as well as to experts wishing to effectively craft software that is *expressive*, *readable* (to themselves and to others), and that supports clear, concise, and maintainable representations of sonic ideas and algorithms.

1.2 “A New Way of Thinking about Audio Programming”

The great computer scientist Alan Perlis once said that “a programming language that doesn’t change the way you think is not worth learning.” Indeed, we are motivated in the design of ChucK to investigate new ways of thinking about programming sound and music, particularly by looking at it from a human-centric perspective (e.g., as opposed to a machine-centric one). As we posited above, a programming language is a highly general and yet highly intimate human-computer interaction (HCI) device.

PL == HCI Device

Figure 1.2: A conjecture.

If that is the case, then perhaps we can think of the task of programming language design as HCI design – loosely speaking. We say “loosely” because while the process embodies the high level principle of designing for humans, we do not necessarily employ any specific theory from the field of human-computer interaction. Sometimes it is the holistic sum of the features, feel, or even “vibe” that can make a programming system appealing, inviting, and ultimately useful. So, much of the design process also tends to be *holistic* in the above sense, which in retrospect for ChucK, remains to be the right decision (we believe). Through this process, we have produced several interesting paradigms and principles that are potentially useful for audio programming, and constructed a practical language that employs these ideas. In addition, the entirety of the programming language and its runtime system present a new way of thinking about developing software for sound synthesis, analysis, composition, live performance, and pedagogy. For example, chapter 4 (On-the-fly Programming) discusses another "equivalence" in the context of writing code live for musical performance and experimentation (see Figure 1.3).

Code == Musical instrument

Figure 1.3: Another conjecture.

1.3 The ChucKian approach

A central tenet of the ChucKian solution to audio programming is to expose programmability/control over time (at vastly different granularities) in cooperation with a time-based concurrent programming model. This gives rise to our notion of a "strongly-timed" audio programming language – one in which the programming has intimate, precise, and modular control over time as it relates to digital audio [95, 96, 99].

In more concrete terms, this entails making time itself both computable and directly controllable, at any granularity. Programmers specify the exact "pattern" with which computation is performed in time by embedded explicit timing information within the code. Based on this semantic, the language's runtime system ensures properties of determinism and precision between the program and time. Furthermore, programmers can specify concurrent code modules, each of which independently controlling their own computations over time but can also be synchronized to other modules via time and other mechanisms (e.g., events and condition variables).

In short, the design of ChucK strives to "hide the mundane aspects of programming, and expose true control". Additionally, ChucK provides an approach for *on-the-fly programming*, where the programmer is enabled and encouraged to develop/test/prototype programs on-the-fly. This style of development has led to applications in prototyping, teaching, and live musical performance where the audience observes the "live code" as musical gestures.

In turn, on-the-fly programming and our interests in exploring ChucK's pedagogical potentials has led to investigations of empowering the programmer, as well as the observers (students, colleagues, audience) through visualization of real-time audio programs and the act of on-the-fly programming. This motivates the Audicle as an integrated development platform that also serves as a real-time program monitor providing feedback to the ChucK programmer [98].

Putting these elements together, this thesis addresses ideas and investigations at the intersection of computer science and music, of technology and art, and of computing and the humans that interact with it.

1.4 Roadmap

In the rest of this document, we explore the history of programming languages for sound/music (Chapter 2). We chronicle the design of ChucK, an audio programming language, and introduce a new way of thinking about music programming, as well as present some of its ramifications (Chapter 3). We discuss the practice of "on-the-fly programming", a new way of rapidly prototyping for experimentation and for live musical performance (Chapter 4). The Audicle, a graphical programming environment for visualizing ChucK in real-time and for aiding on-the-fly programming, is presented in Chapter 5. We then look at the various applications of ChucK in practical contexts, including in performance ensembles such as the Princeton Laptop Orchestra, in classrooms teaching computer science side-by-side with music and sound synthesis/analysis, and in several other arenas (Chapter 6). The conclusion addresses contributions and potential future directions.

Chapter 2

A History of Music and Programming

2.1 Early Eras: Before Computers

The idea of using general-purpose programming computational automata to make music can be traced back to as early as 1843. Ada Lovelace, while working with Charles Babbage, wrote about the applications of the theoretical Analytical Engine, the successor to Babbage's famous Difference Engine. The original Difference Engine was chiefly a "calculating machine" whereas the Analytic Engine (which was never built) was to contain mechanisms for decision and looping, both fundamental to true programmability. Lady Lovelace rightly viewed the Analytical Engine as a general-purpose computer, suited for "developping [sic] and tabulating any function whatever... the engine [is] the material expression of any indefinite function of any degree of generality and complexity." She further predicted the following: "Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."

Lady Lovelace's prediction was made more than a hundred years before the first computer-generated sound. But semi-programmable music-making machines appeared in various forms before the realization of a practical computer. For example, the player piano, popularized in the early 20th century, is an augmented piano that "plays itself" according to rolls of paper (called piano rolls) with perforations representing the patterns to be played. These interchangeable piano rolls can be seen as simple programs that explicitly specify musical scores.

As electronic music evolved, analog synthesizers gained popularity (around the 1960s). They supported interconnecting and interchangeable sound processing modules. There is a level of programmability involved, and this block-based paradigm influenced later design of digital synthesis systems. For the rest of this chapter, however, we are going to focus on programming as specifying computations to make sound and music.

As we step into to the digital age, we divide our discussion into three overlapping eras of programming and programming systems for music. They loosely follow a chronological order, but more importantly each age embodies common themes in how programmers and composers interact with the computer to make sound. Furthermore, we should keep a few overall trends in mind. One crucial trend in this context is that as computers increased in computational power and storage, programming languages tended to become increasingly high-level, abstracting more details of the underlying system. This, as we shall see, greatly impacted the evolution of how we program music.

2.2 The Computer Age (Part I): Early Languages and Rise of MUSIC-N

Our first era of computer-based music programming systems paralleled the age of mainframes (the first generations of "modern" computers in use from 1950 to the late 1970s) and the beginning of personal workstations (mid 1970s). The mainframes were gigantic, often taking up rooms or even entire floors. Early models had no monitors or screens, programs had to be submitted via punch cards, and the results delivered as printouts. Computing resources were severely constrained. It was difficult even to gain access to a mainframe - they were not commodity items and were centralized and available mostly at academic and research institutions (in 1957 the hourly cost to access a mainframe was $200!). Furthermore, the computational speed of these early computers were many orders of magnitude (factors of millions or more) slower than today's machines and were greatly limited in memory (e.g. 192 kilobytes in 1957 compared to gigabytes today) [55, 15]. However, the mainframes were the pioneering computers and the people who used them made the most of their comparatively meager resources. Programs were carefully designed and tuned to yield the highest efficiency.

Sound generation on these machines became a practical reality with the advent of the first digital-to-analog converters (or DAC's), which converted digital audio samples (essentially sequences of numbers) that were generated via computation, to time-varying analog voltages, which can be amplified to drive loudspeakers or be recorded to persistent media (e.g. magnetic tape).

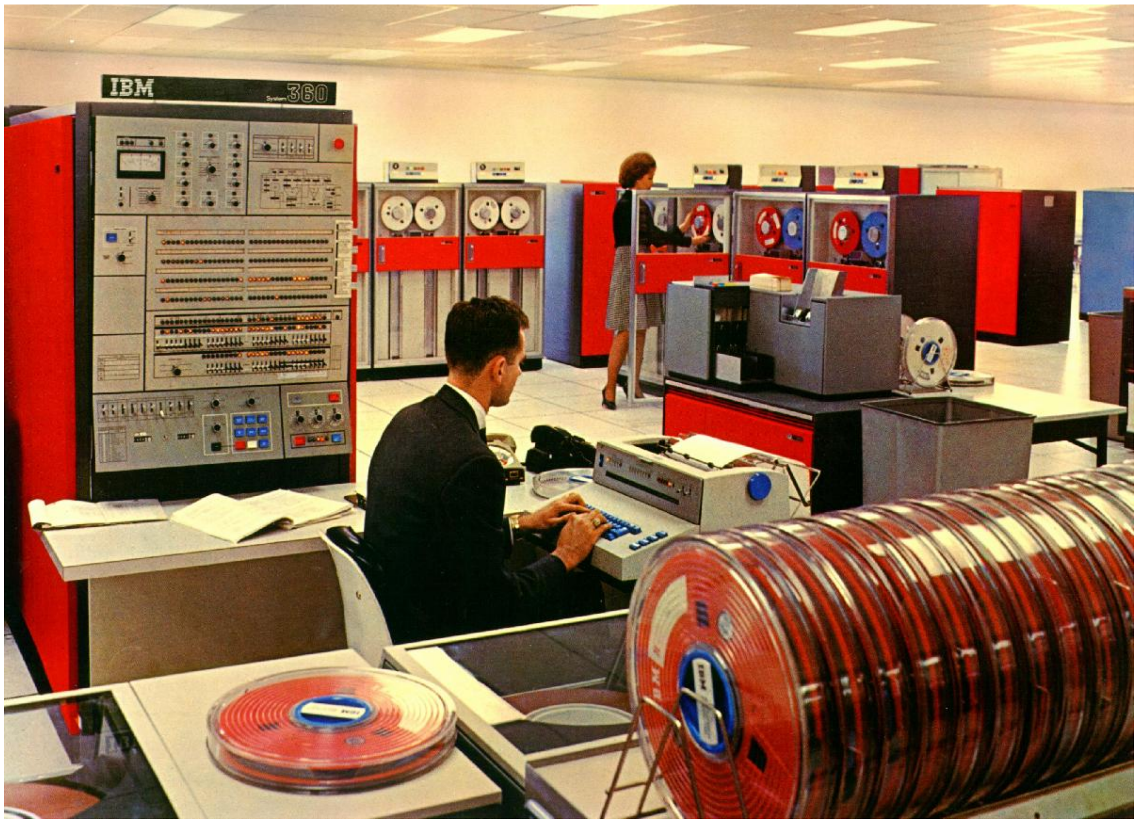


Figure 2.1: The IBM 360, released in 1965, with human operators.

2.2.1 MUSIC I (and II, III, ...)

The earliest programming environment for sound synthesis, called MUSIC, appeared in 1957 [54]. It was not quite a full programming language as we might think of today, but more of an “acoustic compiler”, developed by Max Mathews at AT&T Bell Laboratories. Not only were MUSIC (or MUSIC I, as it was later referred to) and its early descendants the first music programming languages widely adopted by researchers and composers, they also introduced several key concepts and ideas which still directly influence languages and systems today.

MUSIC I and its direct descendants (typically referred to as MUSIC-N languages), at their core, provided a model for specifying sound synthesis modules, their connections, and time-varying control. This model eventually gave rise, in MUSIC III, to the concept of unit generators, or UGen's for short. UGen's are atomic, often predefined, building blocks for generating or processing audio signals. In addition to audio input and/or output, a UGen may support a number of control inputs that control parameters associated with the UGen.

An example of a UGen is an oscillator, which outputs a periodic waveform (e.g. a sinusoid) at a particular fundamental frequency. Such an oscillator might include control inputs that dictate the frequency and phase of the signal being generated. Other examples of UGens include filters, gain amplifiers, and envelope generators. The latter, when triggered, produce amplitude contours over time. If we multiply the output of a sine wave oscillator with that of an envelope generator, we can produce a third audio signal: a sine wave with time-varying amplitude. In connecting these unit generators in an ordered manner, we create a so-called instrument or patch (the term comes from analog synthesizers that may be configured by connecting components using patch cables), which determines the audible qualities (e.g. timbre) of a sound. In MUSIC-N parlance, a collection of instruments is an orchestra. In order to use the orchestra to create music, a programmer could craft a different type of input that contained time-stamped note sequences or control signal changes, called a score. The relationship: the orchestra determines how sounds are generated, whereas the score dictates (to the orchestra) what to play and when. These two ideas - the unit generator, and the notion of an orchestra vs. a score as programs - have been highly influential to the design of music programming systems and, in turn, to how computer music is programmed today (but we get ahead of ourselves).

In those early days, the programming languages themselves were implemented as low-level assembly instructions (essentially human-readable machine code), which effectively coupled a language to the particular hardware platform it was implemented on. As new generations of machines (invariably each with a different set of assembly instructions) were introduced, new languages or at least new implementations had to be created for each architecture. After creating MUSIC I, Max Mathews soon created MUSIC II (for the IBM 740), MUSIC III in 1959 (for the IBM 7094), and MUSIC IV (also for the 7094, but recoded in a new assembly language). Bell Labs shared its source code with computer music researchers at Princeton University - which at the time also housed a 7094 - and many of the additions to MUSIC IV were later released by Godfrey Winham and Hubert Howe as MUSIC IV-B.

Around the same time, John Chowning, then a graduate student at Stanford University, traveled to Bell Labs to meet Max Mathews, who gave Chowning a copy of MUSIC IV. Copy in this instance meant a box containing about 3000 punch cards, along with a note saying “Good luck!”. John Chowning and colleagues were able to get MUSIC IV running on a computer that shared the same storage with a second computer that performed the digital-to-analog conversion. In doing so, they created one of the world’s earliest integrated computer music systems. Several years later, Chowning (who had graduated by then and join the faculty at Stanford), Andrew Moore, and their colleagues completed a rewrite of MUSIC IV, called MUSIC 10 (named after the PDP-10 computer on which it ran), as well as a program called SCORE (which generated note lists for MUSIC 10).

It is worthwhile to pause here and reflect how composers had to work with computers during this period. The composer / programmer would design their software (usually away from the computer), create punch cards specifying the instructions, and submit them as jobs during scheduled mainframe access time (also referred to as batch-processing) - sometimes traveling far to reach the computing facility. The process was extremely time-consuming. A minute of audio might take several hours or more to compute, and turn-around times of several weeks were not uncommon. Furthermore, there was no way to know ahead of time whether the result would sound anything like what was intended. After a job was complete, the generated audio would be stored on computer tape and then be digital-to-analog converted, usually by another computer. Only then could the composer actually hear the result. It would typically take many such iterations to complete a piece of music.

In 1968, MUSIC V broke the mold by being the first computer music programming system to be implemented in FORTRAN, a high-level general-purpose programming language (often considered the first). This meant MUSIC V could be ported to any computer system that ran FORTRAN, which greatly helped both its widespread use in the computer music community and its further development. While MUSIC V was the last and most mature of the Max Mathews / Bell Labs synthesis languages of the era, it endures as possibly the single most influential computer music language. Direct descendants include MUSIC 360 (for the IBM 360) and MUSIC 11 (for the PDP-11) by Barry Vercoe and colleagues at MIT [90], and later cmusic by F. Richard Moore. These and other systems added much syntactic and logical flexibility, but at heart remained true to the principles of MUSIC-N languages: connection of unit generators, and the separate treatment of sound synthesis (orchestra) and musical organization (score). Less obviously, MUSIC V also provided the model for many later computer music programming languages and environments.

2.2.2 The CARL System (or "UNIX for Music")

The 1970s and 80s witnessed sweeping revolutions to the world of computing. The C programming language, one of the most popular in use, was developed in 1972. The 70s was also a decade of maturation for the modern operating system, which includes time-sharing of central resources (e.g. CPU time and memory) by multiple users, the factoring of runtime functionalities between a privileged kernel mode vs. a more protected user mode, as well as clear process boundaries that protect applications from each other. From the ashes of the titanic Multics operating system project arose the simpler and more practical UNIX, with support for multi-tasking of programs, multi-user, inter-process communication, and a sizable collection of small programs that can be invoked and interconnected from a command line prompt. Eventually implemented in the C language, UNIX can be ported with relative ease to any new hardware platform for which there is a C compiler.

Building on the ideas championed by UNIX, F. Richard Moore, Gareth Loy, and others at the Computer Audio Research Laboratory (CARL) at University of California at San Diego developed and distributed an open-source, portable system for signal processing and music synthesis, called the CARL System [52, 63]. Unlike previous computer music systems, CARL was not a single piece of software, but a collection of small, command line programs that could send data to each other. The "distributed" approach was modeled after UNIX and its collection of inter-connectible programs, primarily for text-processing. As in UNIX, a CARL process (a running instance of a program) can send its output to another process via the pipe (|), except instead of text, CARL processes send and receive audio data (as sequences of floating point samples, called floatsam). For example, the command:

```
> wave -waveform sine -frequency 440Hz | spect
```

invokes the wave program, and generates a sine wave at 440Hz, which is then "piped" (|) to the spect program, a spectrum analyzer. In addition to audio data, CARL programs could send side-channel information, which allowed potentially global parameters (such as sample rate) to propagate through the system. Complex tasks could be scripted as sequences of commands. The CARL System was implemented in the C programming language, which ensured a large degree of portability between generations of hardware. Additionally, the CARL framework was straightforward to extend - one could implement a C program that adhered to the CARL application programming interface (or API) in terms of data input/output. The resulting program could then be added to the collection and be available for immediate use.

In a sense, CARL approached the idea of digital music synthesis from a divide-and-conquer perspective. Instead of a monolithic program, it provided a flat hierarchy of small software tools. The system attracted a wide range of composers and computer music researchers who used CARL to write music and contributed to its development. Gareth Loy implemented packages for FFT (Fast Fourier Transform) analysis, reverberation, spatialization, and a music programming language named Player. Richard Moore contributed the cmusic programming language. Mark Dolson contributed programs for phase vocoding, pitch detection, sample-rate conversion, and more. Julius O. Smith developed a package for filter design and a general filter program. Over time, the CARL Software Distribution consisted of over one hundred programs. While the system was modular and flexible for many audio tasks, the architecture was not intended for real-time use. Perhaps mainly for this reason, the CARL System is no longer widely used in its entirety. However, thanks to the portability of C and to the fact CARL was open source, much of the implementation has made its way into countless other digital audio environments and classrooms.

2.2.3 Cmix, CLM, Csound

Around the same time, the popularity and portability of C gave rise to another unique programming system: Paul Lansky's Cmix [67, 46]. Cmix wasn't directly descended from MUSIC-N languages; in fact it's not a programming language, but a C library of useful signal processing and sound manipulation routines, unified by a well-defined API. Lansky authored the initial implementation in the mid-1980s to flexibly mix sound files (hence the name Cmix) at arbitrary points. It was partly intended to alleviate the inflexibility and large turnaround time for synthesis via batch processing. Over time, many more signal processing directives and macros were added. With Cmix, programmers could incorporate sound processing functionalities into their own C programs for sound synthesis. Additionally, a score could be specified in the Cmix scoring language, called MINC (which stands for "MINC Is Not C!"). MINC's syntax resembled that of C and proved to be one of the most powerful scoring tools of the era, due to its support for control structures (such as loops). Cmix is still distributed and widely used today, primarily in the form of RTCmix (the RT stands for real-time), an extension developed by Brad Garton and David Topper [32].

Common Lisp Music (or CLM) is a sound synthesis language written by Bill Schottstaedt at Stanford University in the late 1980s [75]. CLM descends from the MUSIC-N family and employs a Lisp-based syntax for defining the instruments and score and provides a collection of functions that create and manipulate sound. Due to the naturally recursive nature of LisP (which stands for List Processing), many hierarchical musical structures turned out to be straightforward to represent using code. A more recent (and very powerful) LisP-based programming language is Nyquist [24], authored by Roger Dannenberg. (Nyquist is discussed below; both CLM and Nyquist are freely available)

Today, the most widely used direct descendent of MUSIC-N is Csound, originally authored by Barry Vercoe and colleagues at MIT Media Labs in the late 1980s [89, 7, 91]. It supports unit generators as opcodes, objects that generate or process audio. It embraces the instrument vs. score paradigm: the instruments are defined in orchestra (.orc) files, while the score in .sco files. Furthermore, Csound supports the notion of separate audio and control rates. The audio rate (synonymous with sample rate) refers to the rate at which audio samples are processed through the system. On the other hand, control rate dictates how frequently control signals are calculated and propagated through the system. In other words, audio rate (abbreviated as ar in Csound) is associated with sound, whereas control rate (abbreviated as kr) deals with signals that control sound (i.e. changing the center frequency of a resonant filter or the frequency of an oscillator). The audio rate is typically higher (for instance 44100 Hz for CD quality audio) than the control rate, which usually is adjusted to be lower by at least an order of magnitude. The chief reason for this separation is computational efficiency. Audio must be computed sample-for-sample at the desired sample rate. However, for a great majority of synthesis tasks, it makes little perceptual difference if control is asserted at a lower rate, say on the order of 2000Hz. This notion of audio rate vs. control rate is widely adopted across nearly all synthesis systems.

This first era of computer music programming pioneered how composers could interact with the digital computer to specify and generate music. Its mode of working was associated with the difficulties of early mainframes: offline programming, submitting batch jobs, waiting for audio to generate, and transferring to persistent media for playback or preservation. It paralleled developments in computers as well as general-purpose programming languages. We examined the earliest music languages in the MUSIC-N family as well as some direct descendants. It is worth noting that several of the languages discussed in this section have since been augmented with real-time capabilities. In addition to RTCMix, Csound now also supports real-time audio.

2.3 The Computer Age (Part II): Real-time Systems and New Approaches

This second era of computer programming for music partially overlaps with the first. The chief difference is that the mode of interaction moved from offline programming and batch processing to real-time sound synthesis systems, often controlled by external musical controllers. By the early 1980s, computers have become fast enough and small enough to allow workstation desktops to outperform the older, gargantuan mainframes. As personal computers began to proliferate, so did new programming tools and applications for music generation.

2.3.1 Graphical Music Programming: Max/MSP + Pure Data

We now arrive at one of the most popular computer music programming environment to this day: Max and later Max/MSP [68]. Miller S. Puckett implemented the first version of Max (when it was called Patcher) at IRCAM in Paris in the mid-1980s as a programming environment for making interactive computer music. At this stage, the program did not generate or process audio samples; its primary purpose was to provide a graphical representation for routing and manipulating signals for controlling external sound synthesis workstations in real-time. Eventually, Max evolved at IRCAM to take advantage of DSP hardware on NeXT computers (as Max/FTS, FTS stands for "faster than sound"), and was later released in 1990 as a commercial product by Opcode Systems as Max/Opcode. In 1996, Puckette released a completely redesigned and open source environment called Pure Data (PD) [69]. At the time, Pure Data processed audio data whereas Max was primarily designed for control (MIDI). PD's audio signal processing capabilities then made their way into Max as a major add-on called MSP (MSP either stands for Max Signal Processing or for Miller S. Puckett), authored by Dave Zicarelli. Cycling '74, Zicarelli's Company, distributes the current commercial version of Max/MSP. Meanwhile, IRCAM currently maintains jMax [27] as freely available and new implementation of the original Max software.

The modern-day Max/MSP supports a graphical patching environment and a collection containing thousands of objects, ranging from signal generators, to filters, to operators, and user interface elements. Using the Max import API, third party developers can implement external objects as extensions to the environment. Despite its graphical approach, Max descends from MUSIC-V (in fact Max is named after the father of MUSIC-N, Max Mathews) and embodies a similarly modular approach to sound synthesis. A simple Max/MSP example is shown in Figure 2.2.

Max offers two modes of operation. In edit mode, a user can create objects, represented by on-screen boxes containing the object type as well as any initial arguments. An important distinction is made between objects that generate or process audio and control rate objects (the presence of a ˜ at the end of the object name implies audio rate). The user can then interconnect objects by creating connections from the outlets of certain objects to the inlets of others. Depending on its type, an object may support a number of inlets, each of which is well-defined in its interpretation of the incoming signal. Max also provides dozens of additional widgets, including message boxes, sliders, graphs, knobs, buttons, sequencers, and meters. Events can be manually generated by a bang widget. All of these widgets can be connected to and from other objects. When Max is in run mode, the patch topology is fixed and cannot be modified, but the various on-screen widgets can be manipulated interactively. This highlights a wonderful duality: a Max patch is at once a program and (potentially) a graphical user interface.

Max/MSP has been an extremely popular programming environment for real-time synthesis, particularly for building interactive performance systems. Controllers both commodity (MIDI devices) and custom as well as sensors (such as motion tracking) can be mapped to sound synthesis parameters using Max/MSP. The visual aspect of the environment lends itself well to monitoring and fine-tuning patches. Max/MSP can be used to render sequences or scores, though due to the lack of detailed timing constructs (the graphical paradigm is better at representing what than when), this can be less straightforward.

2.3.2 Programming Libraries for Sound Synthesis

So far, we have discussed mostly standalone programming environments, each of which provides a specialized language syntax and semantics. In contrast to such languages or environments, a library provides a set of specialized functionalities for

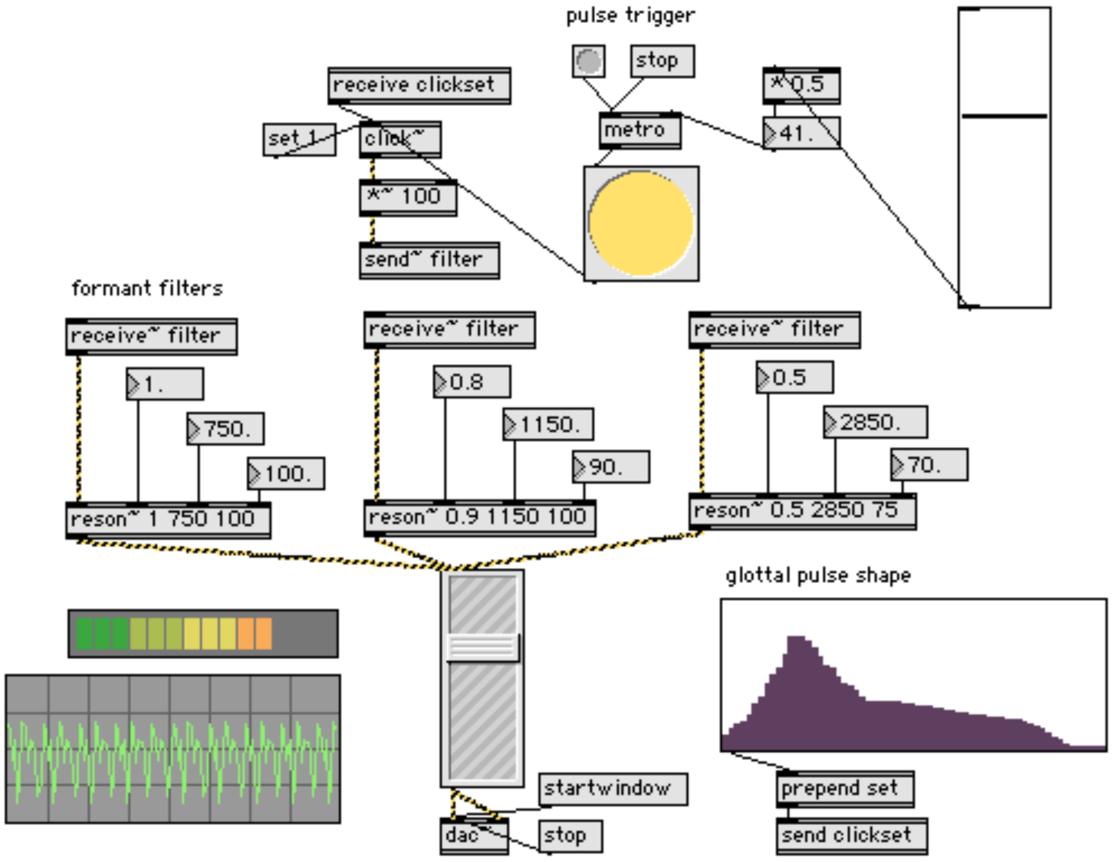


Figure 2.2: A simple Max/MSP patch which synthesizes the vowel ahh.

an existing, possibly more general-purpose language. For example, the Synthesis Toolkit (STK) [20] is a collection of building blocks for real-time sound synthesis and physical modeling, for the C++ programming language. STK was authored and released by Perry Cook in the early 1990's, with later contributions by Bill Butnam and Gary Scavone. JSyn [11], released around the same time, is a collection of real-time sound synthesis objects for the Java programming language. In each case, the library provides an API, with which a programmer can write synthesis programs in the host language (e.g. C++ and Java). For example, STK provides an object definition called Mandolin, which is a physical model for a plucked string instrument.

It defines the data types which internally comprise such an object, as well as publicly accessible functionalities that can be invoked to control the Mandolin's parameters in real-time (e.g. frequency, pluck position, instrument body size, etc.). Using this definition, the programmer can create instances of Mandolin, control their characteristics via code, and generate audio from the Mandolin instances in real-time. While the host languages are not specifically designed for sound, these libraries allow the programmer to take advantage of language features and existing libraries (of which there is a huge variety for C++ and Java). This also allows integration with C++ and Java applications that desire real-time sound synthesis.

2.3.3 Nyquist

Nyquist is an interactive programming language based on Lisp for sound synthesis and music composition [24, 23, 83], and is a culmination of ideas explored in earlier systems such as Arctic [26] and Canon [22]. While adopting familiar elements of audio programming found in earlier MUSIC-N languages, Nyquist (along with SuperCollider, discussed below) is among the first music composition and sound synthesis languages to remove the distinction between the "orchestra" (sound synthesis) and the "score" (musical events): both can be implemented in the same framework. This tighter integration allows both synthesis and musical entities to be specified using a shared "mindset", favoring the high customizability of code over the ease and simplicity of data (e.g., note lists).

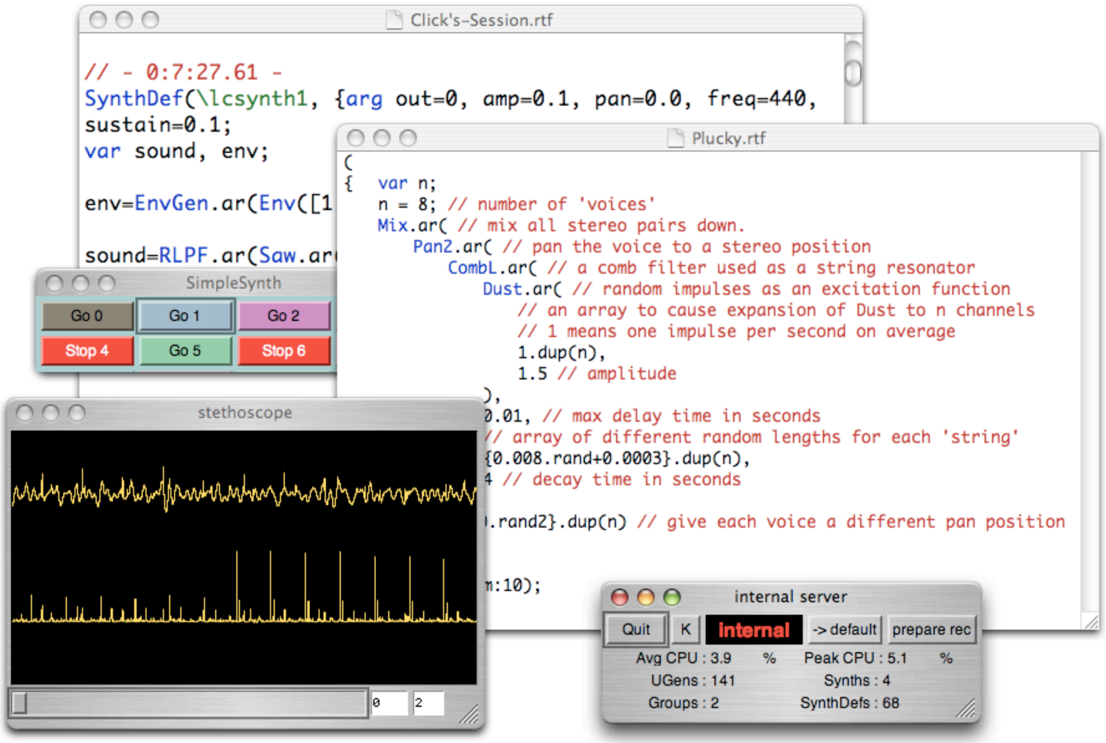
In Nyquist, the composer specifies sound, transformations, and music by combining expressions, leveraging both audio building blocks as well as the full array of features in the general purpose Lisp language and environment. Additionally, Nyquist supports a wide array of advanced ideas. These include "behavioral abstraction", which allows programmers to specify appropriate underlying behaviors in different contexts while maintaining a unified high-level interface. Nyquist also supports the ability to work in both quantitative and perceptual attack times, as well as an advanced abstract time-warping of compound events [23]. At a more basic level, Nyquist offers temporal operators such as **sim** (for simultaneous signals and events) and **seq** (for sequential evaluation).

While Nyquist is not a real-time programming environment (it is interactive), it provides a powerful and intrinsically different way of thinking about programming audio and composing music. Nyquist is in wide use today, including as the primary plug-in programming engine in the open source audio editor Audacity [57].

2.3.4 SuperCollider

SuperCollider is a text-based audio synthesis language and environment [58, 59]. It is highly powerful as a programming language, and the implementation of the synthesis engine is highly optimized. It combines many of the previous ideas in computer music language design while making some fundamental changes and additions. SuperCollider, like languages before it, supports the notion of unit generators for signal processing (audio and control). However, like Nyquist, there is no longer a distinction between the orchestra and score. Furthermore, the language, which in part resembles the Smalltalk and C programming languages, is object-oriented and provides a wide array of expressive programming constructs for sound synthesis and user interface programming. This makes SuperCollider suitable not only for implementing synthesis programs, but also for building large interactive systems for sound synthesis, algorithmic composition, and for audio research.

At the time of this writing, there have been three major version changes in SuperCollider. The third and latest (often abbreviated SC3) makes an explicit distinction between the language (front-end) and synthesis engine (back-end). These loosely coupled components communicate via OpenSoundControl (OSC), a standard for sending control messages for sound over the network. One immediate impact of this new architecture is that programmers can essentially use any front-end language, as long as it conforms to the protocol required by the synthesis server (called scsynth in SuperCollider).



```
// - 0:7:27.61 -
SynthDef(\lcsynth1, {arg out=0, amp=0.1, pan=0.0, freq=440,
sustain=0.1;
var sound, env;

env=EnvGen.ar(Env([1

sound=RLPF.ar(Saw.ar
```

```
(
{   var n;
    n = 8; // number of 'voices'
    Mix.ar( // mix all stereo pairs down.
        Pan2.ar( // pan the voice to a stereo position
            CombL.ar( // a comb filter used as a string resonator
                Dust.ar( // random impulses as an excitation function
                    // an array to cause expansion of Dust to n channels
                    // 1 means one impulse per second on average
                    1.dup(n),
                    1.5 // amplitude
                ),
                0.01, // max delay time in seconds
                // array of different random lengths for each 'string'
                {0.008.rand+0.0003}.dup(n),
                4 // decay time in seconds
            ),
            .rand2}.dup(n) // give each voice a different pan position


n:10);
```

```
Avg CPU : 3.9    %    Peak CPU : 5.1    %
UGens : 141           Synths : 4
Groups : 2            SynthDefs : 68
```

Figure 2.3: SuperCollider programming environment in action.

2.3.5 Graphical vs. Text-based

It is worthwhile to pause here and reflect on the differences between the graphical programming environments of Max/MSP and PD vs. the text-based languages and libraries such as Csound, SuperCollider, STK, and Nyquist (as well as ChucK). The visual representation presents the dataflow directly, in a what-you-see-is-what-you-get sort of way. Text-based systems lack this representation and understanding of the syntax and semantics is required to make sense of the programs. However, many tasks, such as specifying complex logical behavior, are more easily expressed in text-based code.

Ultimately it's important to keep in mind that most synthesis and musical tasks can be implemented in any of these languages. This is the idea of universality: two constructs (or languages) can be considered equivalent if we can emulate the behavior of one using the other, and vice versa. However, certain types of tasks may be more easily specified in a particular language than in others. This brings us back to the idea of the programming language as a tool, and perhaps more importantly, as a *way of thinking*. In general, a tool is useful if it does at least one thing better than any other tool (for example, a hammer or a screwdriver). Computer music programming languages are by necessity more general, but differing paradigms (and syntaxes) lend themselves to different tasks (and no single environment "does it best" in every aspect: it's important to choose the right tools for the tasks at hand). In the end, it's also a matter of personal preference - some like the directness of graphical languages whereas others prefer the feel and expressiveness of text-based code. It's often a combination of choosing the right tool for the task and finding what the programmer is comfortable working in.

2.3.6 Additional Music Programming Languages

Formula, short for *Forth Music Language*, is a programming for computing control signals to synthesizers based on concurrent processes operating in a unified environment that can be scheduled at runtime [3]. Variously processes can be specified to compute pitch sequences as well as control for parameters such as volume, duration, and articulation. Unlike the languages discussed above, Formula computes control signals and does not directly generate or synthesize audio.

Haskore is a set of modules in the Haskell programming language created for expressing musical structures in a high-level declarative style of functional programming. Like Formula, it is more of a language for describing *music* (in Haskore's case, mostly Western music), not *sound* [38]. An advantage of Haskell (and by extension, Haskore) is that objects in the language simultaneously represent abstract (musical) ideas as well as their concrete representation, leading to provable property which can be reasoned about, a result of the programming system.

Formes provides an object-oriented, hierarchical event-based approach to dealing with time [16] and is based on the Lisp programming language. It was not designed to compute audio directly but rather time-oriented control information for the Chant synthesis system.

2.4 The Computer Age (Part III): New Language Explorations

With the growth of low-cost, high performance computers, the real-time and interactive music programming paradigms are more alive than ever and expanding with the continued invention and refinement of new interfaces for musical expression. Alongside the continuing trend of explosive growth in computing power is the desire to find new ways to leverage programming for real-time interaction. If the second era of programming and music evolved from computer becoming commodities, then this third era is the result of programming itself becoming pervasive. With the ubiquity of hardware and the explosion of new high-level general-purpose programming tools (and people willing to use them), more composers and musicians are crafting not only software to create music, but also new software *to create newer and more custom software* for music.

As part of this new age of exploration, a recent movement has been taking shape. This is the rise of dynamic languages and consequently of using the act of programming itself as a musical instrument. This, in a way, can be seen as a subsidiary of real-time interaction, but with respect to programming music, this idea is fundamentally powerful. For the first time in history, we have commodity computing machines that can generate sound and music in real-time (and in abundance) from our program specifications. One of the areas investigated in our third age of programming and music is the possibilities of changing the program itself in real-time as it's running. Given the infinite expressiveness of programming languages, might we not leverage code to create music on-the-fly?

The idea of run-time modification of programs to make music (interchangeably called live coding, on-the-fly programming, interactive programming) is not an entirely new one. As early as the beginning of the 80s, researchers such as Ron Kuivila and groups like the Hub have experimented with runtime modifiable music systems. The Hierarchical Music Scoring Language (HMSL) is a Forth-based language, authored by Larry Polansky, Phil Burk, and others in the 1980s, whose stack-based syntax encourages runtime programming [12]. These are the forerunners of live coding. The fast computers of today enable an additional key component: real-time sound synthesis.

2.4.1 Custom Music Programming Software

An incredibly vibrant and wonderful aspect of the era is the proliferation of custom, "home-brew" sound programming software. The explosion of new high-level, general-purpose, programming platforms has enabled and encouraged programmers and composers to build systems very much tailored to their liking. Alex McLean performs via live coding using the high-level scripting language Perl [60], while developers such as Andrew Sorensen and Andrew Brown have recently explored live coding environments based on Scheme [10]. Similar frameworks have been developed in Python, various dialects of Lisp, Forth, Ruby, and others. Some systems make sound while others visualize it. Many systems send network message (in OpenSoundControl) to synthesis engines such as SuperCollider Server, PD, Max, and ChucK. In this way, musicians and composers can leverage the expressiveness of the front-end language to make music while gaining the functionalities of synthesis languages. Many descriptions of systems and ideas can be found through TOPLAP (which usually stands for Transnational Organisation for the Proliferation of Live Audio Programming), a collective of programmers, composers, and performers exploring live programming to create music [82].

This third era is promising because it enables and encourages new compositional and performance possibilities not only to professional musicians, researchers, and academics, but also to anyone willing to learn and explore programming and music. Indeed, the *homebrew* aesthetic has encouraged personal empowerment and artistic independence from established traditions and trends. Also, the new dynamic environments for programming are changing how we approach more traditional computer music composition by providing more rapid experimentation and more immediate feedback. This era is young but growing rapidly and the possibilities are truly fascinating. Where will it take programming and music in the future?

2.5 Synchronous Reactive Systems

In addition to the realm of audio and music programming, it is worthwhile to provide context for this work with respect to synchronous languages for reactive systems. A *reactive system* maintains an ongoing interaction with its environment, rather than (or in addition to) producing a final value upon termination [35, 34]. Typical examples include air traffic control systems, control kernels for mechanical devices such as wristwatches, trains, and even nuclear power plants. These systems must react to their environment at the environment's speed. They differ from *transformational systems*, which emphasize data computation instead of the ongoing interaction between systems and their environments; and from *interactive systems*, which influence and react to their environments at their own rate (e.g., web browsers).

In synchronous languages for reactive systems, a *synchrony hypothesis* states that computations happen infinitely fast, allowing events to be considered atomic and truly synchronous. This affords a type of logical determinism that is an essential aspect of a reactive system (which should produce the same output at the same points in time, given the same input), and reconciles determinism with the modularity and expressiveness of concurrency. Such determinism can lead to programs that are significantly easier to specify, debug, and analyze compared to non-deterministic ones, for example those written in more "classical" concurrent languages [36, 1].

Several programming languages have embodied this synchrony hypothesis. Esterel [6] is a textual imperative language, suitable for specifying small control-intensive tasks (such as many finite state machines and cases where it's beneficial to emit signals through a system with zero logical delay). In Esterel, the notion of time is replaced by that of *order*, and an Esterel program describes reactions to events that are deterministically and totally ordered in a sequence of *logical instants*. Two events happening at the same logical instant are said to be occurring simultaneously (otherwise, they occur in sequence). Communication in Esterel is carried out exclusively via signals, which can be broadcast with zero delay (i.e., visible to other parts of the system at the same logical instant). Esterel provides a deterministic method to model concurrent low-level control processes and is commonly used in various control systems and embedded devices, being amenable to be implemented in software as well as in hardware.

Other synchronous languages include LUSTRE [13], a declarative dataflow language for reactive systems, and SIGNAL [33]. All of these, including Esterel, are designed for specifying and/or describing low-level reactive systems. As such, they are not "complete" languages - the programs they specify are meant to be integrated into more complex host systems implemented via other means.

2.5.1 ChucK and Synchronous Languages

While it wasn't necessarily designed as such, ChucK possesses aspects of synchronous languages for reactive systems. Computation is assumed to happen infinitely fast with respect to logical ChucKian time, which can only be advanced as explicitly requested and allowed by the program. Compared to the existing synchronous languages for reactive systems, ChucK most closely resembles Esterel - they are both imperative and, each in their own ways, enforce the synchrony hypothesis. Yet, there are some important differences.

Esterel and other synchronous languages are designed for specifying minimal reactive kernels often for integration into more complex host systems. ChucK, on the other hand, is meant to provide a unifying solution for specifying and developing an *entire system*, including a deterministic kernel of control as well as constructs to build complex components and algorithms. It combines high-level, general purpose, audio-centric programmability with the intrinsically low-level benefits offered by the synchrony hypothesis. Additionally, ChucK is highly dynamic, allowing on-the-fly creation of high-level objects and processes.

Existing synchronous languages emphasize *reaction*, whereas ChucK's design goals and programming style are intended to be reactive as well as *proactive* and *interactive*. The ChucK programming model offers events and signal (which are reactive), as well as the ability to specify concurrent processes that move themselves through logical time, to both *control* and to *define* the system. This encourages a fundamentally different, proactive mentality when programming. Additionally, ChucK presents a highly visible and centralized view of logical time (via the **now** keyword) that reconciles logical time with real-time. This mechanism deterministically couples and interleaves user computation with audio computation, providing a continuous notion of time mapped explicitly to the audio stream (see Chapter 3 for discussions, examples, and analysis).

Finally, Esterel is meant to facilitate verification to ensure that the synchrony hypothesis can be reasonably approximated in a practical real-time (often mission-critical) system. ChucK leverages the determinism for program specification, debugging, and analysis, but is less concerned with absolute real-time performance, and more with the determinism bridging code, audio computations, and the ongoing output. For example, even when a ChucK audio synthesis program cannot keep up in real-time on a particular machine, the computed audio samples are still guaranteed to be accurate and interruption-free (e.g., if written to file, and in the absence of asynchronous input). In other words, ChucK can either assume the computer to be infinitely fast, or alternately relax the real-time constraint while maintaining output integrity. In this latter sense, ChucK can also be used in more *transformational* manners.

In this context, ChucK presents a synchronous language that simultaneously embodies elements of reactive, transformational, and interactive systems. Moreover, it embodies a different way of thinking about writing synchronous code that is inextricably related to time and audio. In the upcoming chapters, we shall explore the these and other mechanisms and properties of ChucK.

2.6 Future Directions

What does the future hold for programming and music? As Lady Ada Lovelace foresaw the computing machine as a programming tool for creating precise, arbitrarily complex, and “scientific” music, what might we imagine about the ways music will be made decades and beyond from now?

Several themes and trends pervade the development of programming languages and systems for music. The movement towards increasingly more real-time, dynamic, and networked programming of sound and music continues; it has been taking place in parallel with the proliferation and geometric growth of commodity computing resources until recent times. New trends are emerging. At the time of this writing (between 2006 and 2008), the focus in the design of commodity machines is shifting to distributed, multi-core processing units. We may soon have machines with hundreds (or many more) cores as part of a single computer. How might these potentially massive parallel architectures impact the way we think about and program software (in everything from commercial data-processing to sound synthesis to musical performance)? What new programming paradigms will have to be invented to take advantage of these and other new computing technology such as quantum computers, and (for now) theoretical computing machines? An equally essential question: how can we better make use of the machines we have?

Finally, let's think back to Ada Lovelace's question from the beginning of this chapter, and ponder the following: "Supposing, for instance, that the engine were susceptible of such expression and adaptations, might not the human compose elaborate and scientific pieces of music of any degree of complexity or extent?"

It's always the right time to imagine what new possibilities await us.

Chapter 3

ChucK

3.1 Language Design

The chapter presents the design of the ChucK programming language, its primary features, and its instantiation in the form of the language specification. In these contexts of the design, this chapter then addresses the implementation of the language, as well as some useful properties it offers.

3.1.1 Two Observations

As we formulate the problem statement, we observe two important commonalities that pervade the gamut of audio programming. The first is that *time* is intimately connected with sound and central to how sound and music programs are created and reasoned about. This may seem like an obvious point - as sound and music are intrinsically time-based. Yet we also feel that control over time in programming languages is often under-represented (or sometimes over-abstracted). Low level languages like C/C++ and Java have no inherent notion of time and allow for data types to be built to represent time, which can be cumbersome to implement and use. High level computer music languages tend to abstract time too much, often embodying a more declarative style and connect things in a way that assumes quasi-parallel modules, (e.g., similar to analog synthesizers) while hiding much of the internal scheduling. Timing is also typically broken up into two or more distinct, internally maintained rates (e.g., audio rate and control rate, the latter is often arbitrarily determined for the programmer). The main problem with these existing types of programming model is that the programmer knows *what*, but does not always know *when*.

The second observation is two-fold: 1) sound and music are often the simultaneity of many parallel processes and thus a programming language for music/sound can fundamentally benefit from a concurrent programming model that easily and flexibly captures parallel processes and their interactions. 2) The ability to program parallellism must be both intimately connected to time, and yet it must be provided in such way that it operates *independently* of time. In other words, this functionality must be "orthogonal" to time to provide the maximal degree of freedom and expressiveness.

From these two observations, the ChucKian insight is to expose programmability/control over time (at various granularities) in cooperation with a time-based concurrent programming model. In particular, this entails making time itself both computable and directly controllable at any granularity. Programmers specify the algorithm, logic, or pattern according to which computation is performed in time, by embedded explicit timing information within the code. Based on this framework, the language's runtime system ensures properties of determinism and precision between the program and time. Furthermore, programmers can specify concurrent code modules, each of which can independently control their own computations over time and can also be synchronized to other modules via time and other mechanisms (e.g., events).

3.1.2 Design Goals

Based on the observations in the proceeding section, a set of core language design goals can be summarized as follows:

- **Flexibility**: allow the programmer to naturally express ideas in code, and to flexibly create, edit, and maintain audio programs.
- **Time**: allow the programmer to program the passage of time, and to control and reason about time with precision and across a wide range of temporal granularities.
- **Concurrency**: allow the programmer to write parallel modules that share both data and time, and that can be precisely synchronized; provide a *deterministic* concurrent programming model for audio, minimizing the hassle and complexity of (preemptive) concurrent programming by taking advantage of *time and events* in the language.
- **Readability**: provide/maintain a strong correspondence between code structure and timing.
- **A do-it-yourself language**: combine the expressiveness of lower-level languages and the ease of high-level computer music languages. Support high-level musical concepts, precise low-level timing, and the creation of "white-box" unit generators, all directly in the language.

- **Rapid prototyping**: allow programs to be created and edited as they run, for rapid experimentation, pedagogy, and live performance.
- **Pedagogy**: make audio programming more accessible; an observation is that many people are willing to (learn to) program in order to make music, presenting an opportunity to teach programming more effectively (possibly to people who would otherwise never learn to program). Conversely, the clarity and logic of a programming language can help teach computer music concepts.

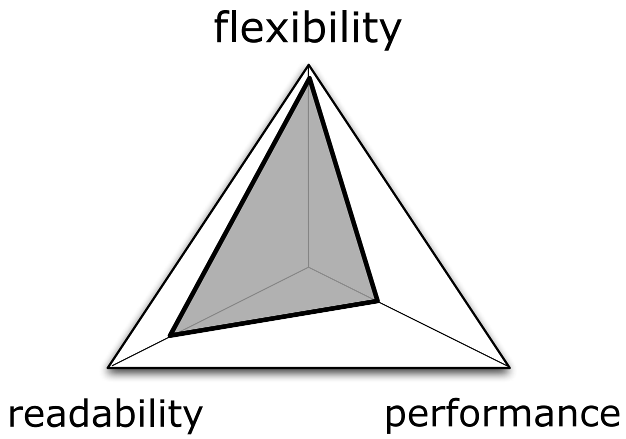


Figure 3.1: Relative emphasis between three design goals.

In terms of the focal points of the design (Figure 3.1), top priority is given to *flexibility* and *readability*. While *performance* (in the sense of computational throughput) is a highly important consideration, it is not our top priority. We design the language to provide maximal control for the programmer, and tailor the system performance around the design.

3.2 Core Language Features

With the design goals outlined above, we present four key ideas that form the foundation of ChucK. The goal is to design a “natural” audio programming language (1) to concurrently and accurately represent complex audio synthesis, (2) to enable fine-grain, flexible control over time, (3) to provide the capability to operate on multiple, dynamic and simultaneous control rates, and (4) to make possible an on-the-fly style of programming. The central ideas are as follows.

- A unifying, massively overloaded operator.
- A precise timing model that unifies high-level and low-level timing and is straightforward to write as well as reason about from code.
- A precise concurrent programming model that supports arbitrarily fine granularity, as well as multiple, simultaneous, and dynamic rates of control.
- A programming paradigm and run-time environment that allow on-the-fly programming, enabling dynamically modifiable programs for performance and experimentation.

3.2.1 ChucK Operator (=>)

ChucK is a strongly-typed, imperative programming language. Its syntax and semantics are governed by a statically-compilable, run-time modifiable type system.

The heart of ChucK’s language syntax is based around the ChucK operator (written as =>). This left-to-right operator, originates from the slang term “to chuck”, meaning to throw an entity into or at another entity. The language uses this notion to help express sequential operations and data flow. => (and related operators) form the “syntactic glue” that binds ChucK code elements together.

=> is a massively overloaded operator, where the behavior of => depends on the context - in particular, what is *being chucked* and what is *chucked to* (see Figure 3.2). In this code fragment, we omit the declaration of the **foo** variable. But assuming we declared **foo** as a unit generator (an audio signal processing element), then the behavior of => in this context would be to connect the output of **foo** into the input of **dac** (another unit generator).

```
// connecting 'foo' to 'dac'
foo => dac;
```

Figure 3.2: A statement using the ChucK operator, here connecting the output of **foo** to **dac** (both are assumed to be unit generators).

A slightly more complex example can be seen in Figure 3.3. This code fragment constructs a simple synthesis instrument using a series of Unit Generators (their declarations are omitted for the moment): a white noise generator, a filter of some type, and the audio output. Notice that the single line captures the flow of the signals from left to right - the same order as ChucK programmers would read and type.

```
// connect 'noise' to 'filter' to 'dac'
noise => filter => dac;
```

Figure 3.3: A statement that uses the ChucK operator to connect three audio elements together.

A ChucK statement can be composed of any appropriate types of objects (including instances of user-defined types), unit generators, operations, values, and variables. The semantic of the statement depends on the types of the objects, and the overloading of the ChucK operator on those types.

In addition to performing connections of Unit Generators, the ChucK operator can be employed in a variety of contexts, including time advancement (see sections on time and concurrency below), function invocation, assignment, and more. For example, Figure 3.4 demonstrates two different syntaxes to achieving the same nested function calls, both valid in ChucK. Note the "un-nesting" effect of using => can lead to more linear and streamlined representations. In this case, the programmer might think of values passing through a sequence of transformations – from left to right.

```
// nested function calls
Math.fabs( Math.min( a, b ) );

// same function calls via =>
( a, b ) => Math.min => Math.fabs;
```

Figure 3.4: Two different syntaxes for invoking the same nested function calls.

Furthermore, there is a greater family of "ChucK operators", ranging from +=> (plus chuck), %=> (modulo chuck), =< (unchuck), to the more recent =^ (upchuck), introduced as part of the ChucK Unit Analyzer framework [101].

3.2.2 ChucKian Time

A key part of the solution in ChucK is to make time itself computable, and also to allow a program to be "self-aware" in the sense that it always knows its position in time, and can control its own progress over time. Furthermore, many program components can share a central notion of *logical ChucKian time*, making it possible to synchronize parallel code solely and naturally based on time as well as to precisely express and reason about the temporal behavior of a program. This gives rise to our notion of a *strongly-timed* audio programming language – one in which programs has intimate, precise, and modular control over their own timing. With respect to synthesis and analysis, an immediate ramification is that control can be asserted at any unit generator at any time and at any rate. In order to make this happen:

- ChucK provides **time** and **dur** as native types in the language (for time and duration).
- The language allows well-defined arithmetic on time and duration (Table 3.1)
- The model provides a deterministic and total mapping of code to time to audio synthesis. It is natural to reason about and specify timing from anywhere in a program.
- The language provides **now**, a special keyword (of type **time**) that holds the current ChucK time. It has a flexible granularity can be orders of magnitude finer than sample-rate, and provides a way to talk about time in an immediate, deterministic, and well-defined sense.
- ChucK offers a globally consistent means to advance time from anywhere in the program flow: by duration (**D +=> now;**) or by absolute time (**T => now;**)

Table 3.1 shows the resulting types of performing various arithmetic on **time** and **dur** types, and whether the operations are commutable by type.

As an example, consider the following program (Figure 3.5), which creates a patch consisting a sine wave generator, and changes its frequency of oscillation randomly every 100 millisecond.

In reading this (or any ChucK) code, it is often helpful follow the code sequential and through the various control structures (e.g., **for/while** loops, **if/else** statements), and noting the points at which ChucKian time is advanced. For example, line 02 instantiates a **SinOsc** (sine wave generator) called **foo**, and connects

type	op	type		result type	commute
`dur`	+	`dur`	$\longrightarrow$	`dur`	✓
`dur`	-	`dur`	$\longrightarrow$	`dur`	
`dur`	*	`float`	$\longrightarrow$	`dur`	✓
`dur`	/	`float`	$\longrightarrow$	`dur`	
`dur`	/	`dur`	$\longrightarrow$	`float`	
`time`	+	`dur`	$\longrightarrow$	`time`	✓
`time`	-	`dur`	$\longrightarrow$	`time`	
`time`	-	`time`	$\longrightarrow$	`dur`	

Table 3.1: Arithmetic operations on `time`, `dur`, and `float` types, and the resulting types.

```
#01 // synthesis patch
#02 SinOsc foo => dac;
#03
#04 // infinite time loop
#05 while( true )
#06 {
#07     // randomly choose a frequency
#08     Std.rand2f( 30, 1000 ) => foo.freq;
#09     // advance time
#10     100::ms => now;
#11 }
```

Figure 3.5: A ChucK program to generate a sine wave, changing its frequency of oscillation every 100 milliseconds.

it to the **dac** (abstraction for audio output). The program flow enters the **while** loop on line 05, randomly chooses a frequency between 30 and 1000 Hz for the sine wave (line 08). The program advances time by 100 milliseconds on line 10, before returning to check the loop conditional again on line 05. It is important at this point to understand what really happens on line 10. By *chucking* the duration **100::ms** to the special ChucKian time variable **now**, the program flow *pauses* and returns control to the ChucK virtual machine and synthesis engine, which generates audio corresponding to 100 milliseconds (precisely, to the nearest sample), before returning control back to our program. In this sense, advancing time in ChucK is similar to a **sleep** call found in many languages. The difference is that here the language *synchronously guarantees* precision in logical time (which can be mapped to the nearest audio sample), allowing one to specify complex timing process across the system. Furthermore, the same method of reading the code can be applied to more complex ChucK programs, to reason about the timing in a straightforward way.

The code above describes one of two ways to "advance time" in ChucK. In the first method (*chuck-to-now*) the programmer can allow time to advance by explicitly chucking a duration value or a time value to **now**, as shown above. This allows for a natural programming approach that embeds the timing control directly in the language, giving the programmer the ability to perform computations at arbitrary points in time, and to "move forward" in ChucK time in a precise manner. The second method to advance time in ChucK is by waiting on one or more event(s). An event could represent synchronous software triggers, as well as asynchronous message over MIDI, OpenSoundControl [104], and HID input devices. User code execution will resume when the synchronization condition is fulfilled; while the event is waited upon, the virtual machine is free to schedule audio synthesis and other synchronous computations. *Wait-on-event* is similar in spirit to *chuck-to-now*, except events have no pre-computed time of arrival.

It is essential to point out that the logical ChucKian time *stands still* until explicitly instructed to do so by one of the above statements to move forward in time. This allows an arbitrary amount of computation to be specified at any single point in time, and makes the synchrony assumption that computation happens infinitely fast. By combining this abstraction with the mapping of ChucK time to the audio sample stream, ChucK's timing mechanisms provides a way for programmers to reason about about time in their audio code in a truly precise and deterministic way.

Another important point to note is that all synthesis systems, at some level, have to be *sample-synchronous* (samples precisely synchronized with time) – or else DSP just doesn't happen. The question to ask here is: at what level does *the language* expose the ability to control timing? In ChucK, precise control over time is designed to be available at all levels to the programmer, in a synchronous manner.

In summary, the timing mechanism moves the primary control over time from inside opaque unit generators to the code directly. The result is that the computation is explicitly tied to time. The programmer not only knows *what* is to be computed, but also precisely *when*, relative to ChucK time. This global control over time enables programs to specify arbitrarily complex timing, allowing a programmer / composer to "sculpt" a sound or passage into perfection by operating on it at any temporal granularity.

We are not the first to address this issue of enabling low-level timing in a high-level audio programming language. Music V, HMSL, and Nyquist have all embodied some form of sample-synchronous programming model. More recently, Chronic [8, 9], with its temporal type constructors, was designed to make arbitrary sub-control rate timing programmable for synthesis. While the mechanisms of Chronic are very different from ChucK's, one aim is the same: to free programmers from having to implement "black-box" unit generators (in a lower-level language, such as C/C++) when a new lower-level feature is desired. In a sense, Chronic "zooms out" and deals time in a global, non-real-time way. On the other hand, ChucK "zooms in" and operates at a specific point in time, in an highly immediate manner.

Thus far, we have discussed programming ChucK using one path of execution, controlling it *through time*. However, time alone is not enough, since audio and music are the simultaneity of potentially many parallel processes. We also desire concurrency in order to expressively capture parallelism. ChucK is a concurrent programming language, and allows multiple independent paths of computation to be executed in parallel. The flexibility and power of the timing mechanism is greatly extended by ChucKs concurrency model, which allows multiple, precisely timed paths of computation.

3.2.3 Concurrency Based on Shreds

The ChucK programming language natively enables the chuckist to write code that operates either in series or in parallel via ChucK's concurrency model. It is also this mechanism that provides fine-grain, multiple, and simultaneous control rates. To this end, ChucK introduces a primitive called *shreds*. A shred, much like a thread, is an independent, lightweight process, which operates concurrently and can share data with other shreds. However, unlike traditional threads, whose execution is interleaved in a non-deterministic manner by a preemptive scheduler, a shred is a deterministic piece of computation that has sample-accurate control over audio timing, and is naturally synchronized with all other shreds via the same timing mechanism.

ChucK shreds are programmed in much the same spirit that traditional threads are, with the exception of several key differences:

- A ChucK shred cannot be preempted by another. This not only enables a single shred to be locally deterministic, but also an entire set of shreds to be globally deterministic in their timing and order of execution.

- A ChucK shred must voluntarily relinquish the processor for other shreds to run (In this they are like non-preemptive threads). But it does not do so with a **yield()**-like function found in many concurrent systems. Shreds, by design, directly use ChucK's timing mechanism: when a shred advances time or waits for an event, it is, in effect, *shreduled* by the *shreduler* (which interacts with the audio engine), and relinquishes the processor. This is powerful in that it can naturally synchronize shreds to each other by time, without using any traditional synchronization primitives.

- ChucK shreds are implemented completely as user-level primitives. The entire virtual machine runs in user-space. User-level parallelism has significant performance benefits over kernel threads [4], allowing "even fine-grain processes to achieve good performance if the cost of creation and managing parallelism is low." Indeed, ChucK shreds are lightweight - each only contains minimal state. The cost of context switching between ChucK shreds is also low since no kernel interaction is required. Furthermore, a user-level shreduler is more readily modifiable.

An advantage of the shred approach is that the programmer has complete control over timing and the interaction of shreds. By leveraging concurrency infused with the deterministic timing mechanism, we gain the benefit of concurrent programming without many of its hassles (e.g., deadlock, race conditions, etc.). We also gain the performance advantages from user-level parallelism. Furthermore, real-time scheduling optimizations [21] can be readily implemented by the shreduler without any kernel modifications. One potential drawback is that a single shred could hang the ChucK virtual machine (along with all other active shreds), if it fails to relinquish the processor. However, there are ways to alleviate this drawback. For example, any hanging shreds can easily by identified by the ChucK Virtual Machine (it would be currently running shred), and the ChucK timing semantic makes it straightforward for the programmer to locate and correct such issues. For example, on-the-fly programming allows for hanging shreds to be removed and corrected during run-time without stopping or restarting the system.

Multi-shredded programs, while no more computationally powerful than single-shredded programs (it should be possible to implement one in terms of the other), can make the task of managing concurrency and timing much easier (and more enjoyable), just as threads make concurrent programming manageable, and potentially increase throughput. In this sense, shreds are more powerful programming constructs. We argue that the flexibility of shreds to empower the programmer to do deterministic, precisely timed, concurrent audio programming significantly outweighs the potential drawbacks.

In a high level sense, the idea of concurrency in ChucK is similar to the idea of mixing independent "tracks" of audio samples in CMix [46] (and other languages). Lansky's original idea was to provide a programming environment where the composer can deal with and perfect individual parts independently [67]. ChucK extends this idea by allowing full programmability for each shred.

Aside from asynchronous input events (e.g., incoming HID, MIDI, OSC messages), a ChucK program is completely deterministic in nature - there is no preemptive background processing, nor any implicit scheduling. The order that shreds and the virtual machine subsystem executes are completely determined by the timing and synchronization specified in the shreds. This makes it easy to reason about the global sequence of operations and timing in ChucK. The concurrency model also enables multiple shreds to run at arbitrary control rates.

This yields a programming model in which multiple concurrent shreds synchronously construct and control a global unit generator network over time. The shreduler uses the timing information to serialize the shreds and the audio computation in a globally synchronous manner. It is completely deterministic (real-time input aside) and the synthesized audio is guaranteed to be correct, even when real-time isn't feasible.

3.2.4 Synthesis and Analysis

In tandem with audio synthesis, ChucK also provides the means to specify and perform precisely-timed and concurrent audio analysis. This both leverages the synthesis framework and extend it with a set of syntactic operators and semantics specifically tailored for analysis [101]. At this time of this writing, we've only begun to explore the possibilities. (See Unit Analyzers in the Language Specification section, as well as the Future Work section in the Conclusions chapter).

3.3 Language Specification

The section describes salient language specifications of ChucK, and provides an example-based view of programming in ChucK.

3.3.1 Types, Values, Variables

ChucK is a strongly-typed language, in which types are resolved at compile-time. However, it is not quite *statically-typed*, because the compiler/type system is a part of the ChucK virtual machine, and is a runtime component. The type system helps to impose precision and clarity in the code, and naturally lends to organization of complex programs. At the same time, it is also dynamic in that changes to the type system can take place in a well-defined manner at runtime. This dynamic aspect also helps to form the basis for on-the-fly programming.

As in other strongly-typed programming languages, we can think of a type as associated behaviors of data. (For example, an **int** is a type that means integer, and adding two integers is defined to produce a third integer representing the sum.) Classes and objects allow us to extend the type system with our own custom types, but we won't cover them here. We mainly focus on primitive types here, and leave the discussion of more complex types for classes and objects to the full language specification [92].

Primitive Types

Primitive types are simple datatypes (with no additional data attributes). By contrast, Objects are not primitive types. Primitive types are passed by value, and cannot be extended. The primitive types in ChucK are:

- **int** : integer (signed)
- **float** : floating point value (by default double-precision)
- **time** : ChucKian time
- **dur** : ChucKian duration
- **void** : (no type)
- **complex** : complex number in rectangular form (see below)
- **polar** : complex number in polar form (see below)

All other types are derived from **Object**, either as part of the ChucK standard library, or as new custom classes.

Reference Types

Reference types are those that inherit from the **Object** class. Some default reference types include:

- **Object** : base type that all classes inherit from (directly or indirectly)
- **(array)** : N-dimensional ordered set of data
- **Event** : fundamental, extendable, synchronization mechanism
- **UGen** : extendable unit generator base class
- **UAna** : extendable unit analyzer base class (inherits UGen)

ChucK supports the ability extend the type system with additional classes, and via *polymorphic* inheritance.

Complex Types

Two special primitive types are available to to represent complex data, such as the output of an FFT: complex and polar. A complex number of the form a + bi can be declared, where the #(...) syntax explicitly denotes a complex number in rectangular form, and can be used in arithmetic calculations. The (floating point) real and imaginary parts of a complex number can be accessed with the **.re** and **.im** components of a complex number. (Figure 3.6).

```
#(5,-1.5) => complex cmp; // 5 - 1.5i
#(2,3) + #(5,6) + cmp => complex sum; // 12 + 7.5i

#(2.0,3.5) => complex cmp;
cmp.re => float x; // x is 2.0
cmp.im => float y; // y is 3.5
```

Figure 3.6: complex values, real/imaginary components.

The polar type offers an equivalent, alternative representation of complex numbers in terms of a magnitude and phase value. The magnitude and phase values can be accessed via **.mag** and **.phase**. (Figure 3.7)

```
%(2,.5*pi) => polar pol; // polar
pol.mag => float m; // m is 2
pol.phase => float p; // p is .5*pi
```

Figure 3.7: polar values, magnitude/phase components.

Polar and complex representations can be cast to each other and used in arithmetic operations. (Figure 3.8).

```
// polar
%(2,.5*pi) => polar pol;
// complex
#(3,4) => complex cmp;
// casting
pol $ complex + #(10,3) + cmp => complex cmp2;
// casting
cmp $ polar + %(10,.25*pi) - pol => polar pol2;
```

Figure 3.8: Some operations on complex and polar types in ChucK.

3.3.2 Arrays

Arrays are used to hold N-dimensional, ordered sets of data (of the same base type). Some notes on ChucK arrays can be found below.

- arrays can be indexed by integer (0-indexed).
- any array can also be used as an associative map, indexed by strings.
- it is important to note that the integer-indexed portion and the associative portion of an array are stored in separate namespaces.

- arrays are objects (see objects and classes), and will behave similarly under reference assignment and other operations common to objects.

3.3.3 Operators

In addition to the ChucK operator (=>), ChucK offers standard operators for arithmetic, binary, and logical operations, there exists a family of *extended ChucK operators*, including the *upchuck* (=^, see Unit Analyzers), the *unchuck* (=<), and a variety of *operate-and-assign* operators (e.g., +=> for incrementing a variable by an amount with assignment).

3.3.4 Control Structures

ChucK employs many standard control structures found in procedural programming languages, including **if**, **else**, **for**, **while**, and introducing additional control structures such as **until** (the semantic opposite of **while**), and **repeat**, which evaluates the control expression as an integer only once and repeats the body that number of times.

3.3.5 Manipulating Time

Notions of time and concurrency are central to understanding and working with ChucK. The main points of time in ChucK are summarized below.

- time and duration are native types in the language.
- the **now** keyword holds the current logical ChucK time.
- time is advanced (and can only be advanced) by explicitly manipulating **now**.

In ChucK. **time** represents an absolute point in time (from the beginning of ChucK time), and **dur** represents a duration (with the same logical units as time). See Figure 3.9 for some basic examples.

```
// duration of one second
1::second => dur foo;

// a point in time (duration 'foo' after 'now')
now + foo => time later;
```

Figure 3.9: Examples of basic operations on **time** and **dur**.

Durations can be used to construct new durations, which themselves can be used to inductively construct yet other durations. For example, see Figure 3.10.

```
// .5 second is a quarter
.5::second => dur quarter;

// 4 quarters is whole
4::quarter => dur whole;
```

Figure 3.10: Example of constructing the notions of a **quarter** and a **whole** with durations.

By default, ChucK provides these preset duration values:

- **samp** : duration of 1 sample in ChucK time
- **ms** : 1 millisecond
- **second** : 1 second
- **minute** : 1 minute
- **hour** : 1 hour
- **day** : 1 day
- **week** : 1 week

These can be used to represent a wide range of durations and temporal granularities. See Figure 3.11 for some examples of using durations.

```
// the duration of half a sample
.5::samp => dur foo;

// 20 weeks
20::week => dur waithere;

// use in combination
2::minute + 30::second => dur bar;

// same value as above
2.5::minute => dur bar2;
```

Figure 3.11: Some examples of using the **dur** type.

Advancing Time

Advancing time allows other shreds (processes) to run and allows audio to be computed in a controlled manner. There are three ways of advancing time in ChucK:

- chucking (+=>, or with the more common shorthand, =>) a duration to **now**: this will advance time by that duration. (the duration must be nonnegative)
- chucking (=>) a time to **now**: this will advance time to that point. (note that the desired time must be later than or equal to the current time)
- chucking (=>) an event to **now**: time will advance until the event is triggered. (also see the section *Programming with Events*)

See Figure 3.12 for examples of advancing time by *chucking* duration values to **now**.

A time chucked to **now** will have ChucK wait until the appointed time. In the logical, synchronous sense, ChucK should never miss an appointment. Again, the

```
// advance time by 1 second
1::second => now;

// advance time by 100 millisecond
100::ms => now;

// advance time by 1 samp (duration of a sample)
1::samp => now;

// advance time by less than 1 samp
.024::samp => now;
```

Figure 3.12: Some examples of advancing time with durations in ChucK.

destination time *chucked* to **now** must be greater than or equal to **now**, otherwise an exception is thrown. Figure 3.13 shows an example of time and duration in action.

```
// compute value that represents "5 seconds after now"
now + 5::second => time later;

// while we are not at later yet...
while( now < later )
{
    // print out value of now
    <<< now >>>;

    // advance time by 1 second
    1::second => now;
}
```

Figure 3.13: A short program demonstrating **time** and **dur** types.

Figure 3.14 shows another simple example; note how one might follow the code from top to bottom and through the control structures, much like how control flows as the computer executes the code.

Furthermore, there are no restrictions (other than underlying floating point precision) on how much time is advanced. So it is possible to advance time, say, by

```
// our patch: sine oscillator to dac
SinOsc s => dac;

// infinite time loop
while( true )
{
    // randomly choose frequency from 30 to 1000
    Std.rand2f( 30, 1000 ) => s.freq;

    // advance time by 100 millisecond
    100::ms => now;
}
```

Figure 3.14: A sound generating program that randomizes frequencies every 100 milliseconds.

a microsecond, a samp, 2 hours, or 10 years. The system will behave accordingly and deterministically. This mechanism allows time to be controlled at any desired rate, according to any programmable pattern. With respect to audio programming, it is possible to control any unit generator (or unit analyzer) at *any* rate, even sub-sample rate.

Alternately, time can be advanced by chucking an event to **now**, the difference being that with events, the programmer does not specify the duration to wait ahead of time, but rather allows the ChucK virtual machine to move forward in time, while waiting for a particular type of event to occur. See the Event subsection below for more details.

3.3.6 Functions

Functions in ChucK are similar to those found in other procedural programming languages, such as Java, C, and C++, and will not be discussed here. Again, note that => can be used to *un-nest* function calls (refer back to Figure 3.4 to see an example). For more details see the full current language specification [92].

3.3.7 Concurrency, Processes, and Shreds

ChucK is able to run many processes concurrently (the process behave as if they are running in parallel). As mentioned above, a ChucKian process is called a *shred*. To *spork* a shred means creating and adding a new process to the virtual machine. Shreds may be sporked from a variety of places, and may themselves spork new shreds.

ChucK supports synchronous, non-preemptive concurrency. Any number of programs/shreds can be automatically shreduled and synchronized using the programmer-specified timing directives (e.g., chucking a duration to **now**). The concurrency is *sample-synchronous*, meaning that inter-process audio timing is guaranteed to be precise to the sample. Note that any given shred does not necessarily need to know about other shreds - it only has to deal with time locally. The virtual machine ensures that computations happen correctly across the system. Like timing, concurrency is deterministic in ChucK.

Sporking Shreds

The simplest way to to run shreds concurrently is to specify them on the command line, running any number of chuck programs in sample-synchronous concurrency:
%> **chuck foo.ck bar.ck boo.ck**

New ChucK shreds can also be sporked from within ChucK programs. To spork a shred from code, use the **spork ~** operator, which has the following properties:

- the **spork** keyword dynamically sporks shred from within a function call.
- this operation is synchronous, the new shred is shreduled to execute immediately in logical time, starting at the sporked function.

- the parent shred continues to execute, until time is advanced or until it explicitly yields.
- when a parent shred exits, all child shreds also exit.
- sporking a function returns a reference to the new shred.

Figure 3.15 shows an example that defines a function, and then sporks a new shred to start executing at the defined function. A slightly more involved example can be found in Figure 3.16.

```
// define function go()
fun void go()
{
    // (code goes here)
}

// spork a new shred to start running from go()
spork ~ go();

// spork another, store reference to new shred
spork ~ go() => Shred @ offspring;

// ... more code
```

Figure 3.15: Defining a function, then sporking that function on a new shred.

The ’me’ keyword

The **me** keyword (of type **Shred**) refers to the current shred. Basic functions common to all shreds include **yield()**, **exit()**, and **id()**.

For example, it is sometimes useful to suspend the current shred without advancing time, and run others shreds shreduled at the current time. **me.yield()** achieves exactly that. This is often useful for allowing a newly sporked shred to

```
// define function
fun void foo( string s )
{
    // infinite time loop
    while( true )
    {
        // print s
        <<< s >>>;
        // advance time
        500::ms => now;
    }
}

// spork shred, passing in "you" as argument
spork ~ foo( "you" );
// advance time by 250 ms
250::ms => now;
// spork another shred
spork ~ foo( "me" );

// infinite time loop - to keep child shreds around
while( true ) 1::second => now;
```

Figure 3.16: Defining a function, sporking two copies of it on new shreds..

have chance to run without advancing time (Figure 3.17). The programmer can get a shred's ID number via **id()** (Figure 3.18).

It may also be useful to exit the current shred. This can be done by invoking **me.exit()**. For example if a MIDI device fails to open, the programmer might exit the current shred (Figure 3.19).

Machine.add(string path) takes the path to a ChucK program and sporks it. Unlike **spork ˜**, there is no parent-child relationship between the shred that calls the function and the new shred that is added. This is useful for dynamically running stored programs. Similarly, one can remove shreds from the virtual machine. (Figure 3.20)

Shreds sporked in the same file can share the same global variables. They can

```
// spork shred
spork ~ go();

// suspend the current shred...
// (give other shreds a chance to run at 'now')
me.yield();
```

Figure 3.17: Example showing the **yield()** function, which relinquishes the VM without advancing time.

```
// print out the shred id
<<< me.id(); >>>;
```

Figure 3.18: Prints out the current shred's id.

use time and events to synchronize to each other. Shreds sporked from different files can share data (including events). This can be done through a public class with public static variables and functions.

ChucK supports passing arbitrary data from the command line into ChucK programs using optional command line arguments. An argument is specified by appending a colon character (**:**) to the name of the ChucK program to receive that argument, followed by the argument list. Multiple arguments can be specified, each separated by the colon character. Furthermore, each ChucK program can have its own set of arguments. Command line arguments can also be used when using on-the-fly programming facilities of ChucK. **Machine.add()** and **Machine.replace()** accept command line arguments in a similar fashion (Figure 3.21).

To access command line arguments within a ChucK program, one can invoke the **me.args()** and **me.arg()** functions from the shred to determine the size and the contents of the argument list(Figure 3.22).

```
// make a MidiIn object
MidiIn min;

// try to open device 0
if( !min.open( 0 ) )
{
    // print error message
    <<< "can't open MIDI device..." >>>;
    // exit the current shred
    me.exit();
}
```

Figure 3.19: Attempts to open a MIDI device, and exits if the operation fails.

```
// add
Machine.add( "foo.ck" ) => int id;

// remove shred with id
Machine.remove( id );

// add
Machine.add( "boo.ck" ) => id;

// replace shred with "bar.ck"
Machine.replace( id, "bar.ck" );
```

Figure 3.20: Operations using static members of the **Machine** class.

```
// run foo; pass "1" and "bar" as command line arguments
Machine.add( "foo.ck:1:bar" ) => int id;

// replace shred with "bar.ck"
// pass "2" and "boo" as command line arguments
Machine.replace( id, "bar.ck:2:boo" );
```

Figure 3.21: Passing arguments to shreds via **Machine**.

```
// print out all arguments
for( int i; i < me.args(); i++ )
    <<< me.arg( i ) >>>;
```

Figure 3.22: Loops through shred arguments and prints each.

3.3.8 Programming with Events

ChucK events are derived from the native **Event** class within the ChucK language. One can instantiate or otherwise obtain an event, and chuck (=>) that event to **now** to synchronize on it. At this point, the event places the current shred on the event's waiting list and suspends the shred (letting time advance from that shred's point of view). When the event is triggered, one or more of the shreds on its waiting list is shreduled to run immediately. This trigger may originate from another ChucK shred, or from activities taking place outside the Virtual Machine (e.g., from MIDI, OSC, or HID). A simple example is shown in Figure 3.23.

```
// instantiate an Event
Event e;

// ... (other code possibly)

// wait on the event (to be trigger from elsewhere)
e => now;

// after the event is triggered, print message
<<< "I just woke up" >>>;
```

Figure 3.23: Code snippet to wait on an Event, and printing a debug message.

Events can be triggered in one of two ways, depending on the desired behavior. **signal()** releases the first shred in that event's queue and shredules it to run at the current time, respecting the order in which shreds were added to the queue. By contrast, **broadcast()** releases and shredules all shreds queued on the event, in the order they were added, and at the same instant in ChucK time.

The released shreds are shreduled to run immediately, subject to other shreds also shreduled to run at the same time. Furthermore, the shred that called **signal()** or **broadcast()** will continue to run until it advances time itself, or yields the virtual machine without advancing time. For a program that demonstrates **signal()** and **broadcast()**, see Figure 3.24.

MIDI Events

ChucK contains built-in MIDI classes to allow for interaction with MIDI based software or devices. **MidiIn** is a subclass of **Event**, and as such can be chucked to **now** for synchronization. Upon the arrival of incoming MIDI messages, the MidiIn instance wakes up the waiting shred and uses a **MidiMsg** object to return the data.

OSC Events

In addition to MIDI, ChucK provides OpenSoundControl (OSC) communication classes as well [104] (Figure 3.26). The **OscRecv** class listens for incoming OSC packets on the specified port. Each instance of OscRecv can create **OscEvent** objects using its **event()** method to listen for packets at any valid OSC address pattern. An OscEvent object can then be chucked to **now** to wait for new messages to arrive, after which the **nextMsg()** and **getFloat/String/Int()** methods can be used to fetch message data.

In addition to MIDI and OSC, ChucK also supports HID's (e.g., human interface devices, such as computer keyboards, mice, and game controllers). The semantics of HID's are also event-based and analogous to that of MIDI and OSC, and will not be discussed further here.

Lastly, events, like most other classes, can be subclassed to add functionality; see Figure 3.27 for an example of this.

```
// declare event
Event e;

// function for shred
fun void eventshred( Event event, string msg )
{
    // infinite loop
    while ( true )
    {
        // wait on event
        event => now;
        // print
        <<< msg >>>;
    }
}

// create shreds
spork ~ eventshred ( e, "fee" );
spork ~ eventshred ( e, "fi" );
spork ~ eventshred ( e, "fo" );
spork ~ eventshred ( e, "fum" );

// infinite time loop
while ( true )
{
    // either signal or broadcast
    if( maybe )
    {
        <<< "signaling..." >>>;
        e.signal();
    }
    else
    {
        <<< "broadcasting..." >>>;
        e.broadcast();
    }

    // advance time
    0.5::second => now;
}
```

Figure 3.24: An example that sporks four shreds, and invokes them via **signal()** and **broadcast()**.

```
// instantiate MIDI related objects
MidiIn min;
MidiMsg msg;

// open midi receiver, exit on fail
if ( !min.open(0) ) me.exit();

// loop
while( true )
{
    // wait on midi event
    min => now;

    // receive midimsg(s)
    while( min.recv( msg ) )
    {
        // print content
        <<< msg.data1, msg.data2, msg.data3 >>>;
    }
}
```

Figure 3.25: A program to open a MIDI device, wait on incoming messages, and print them.

3.3.9 Unit Generators

Unit generators (**UGen**'s) are function generators that output signals that can be used as audio or control signals. In ChucK, where there is no fixed control rate, any unit generator may be controlled at any rate. Using the timing mechanism, one can program one's own control rate, and can dynamically vary the rate over time. Moreover, ChucK's concurrent programming model make it possible to specify many different parallel controls rates, each at its desired granularity. Some additional properties of ChucK unit generators are listed below.

- All ChucK unit generators are objects.
- All ChucK unit generators inherit from the **UGen** class.

```
// patch
SinOsc foo => dac;

// create our OSC receiver
OscRecv recv;
// port 6449
6449 => recv.port;
// start listening (launch thread)
recv.listen();

fun void rateShred()
{
    // create an address in the receiver
    // and store it in a new variable
    recv.event("/synth/control/freq, f") @=> OscEvent freqEvent;

    // loop
    while ( true )
    {
        // wait for events to arrive
        freqEvent => now;

        // grab the next message from the queue
        while( freqEvent.nextMsg() != 0 )
        {
            // getFloat fetches the expected float
            // as indicated in the type string "f"
            freqEvent.getFloat() => foo.freq;
        }
    }
}
```

Figure 3.26: A OpenSoundControl receiver.

```
// extended event
class TheEvent extends Event
{
    int value;
}

// the event
TheEvent e;

// handler
fun int hi( TheEvent event )
{
    while( true )
    {
        // wait on event
        event => now;
        // get the data
        <<< e.value >>>;
    }
}

// spork 4 copies
spork ~ hi( e );
spork ~ hi( e );
spork ~ hi( e );
spork ~ hi( e );

// infinite time loop
while( true )
{
    // set data
    Math.rand2( 0, 5 ) => e.value;
    // signal one waiting shred
    e.signal();
    // advance time
    1::second => now;
}
```

Figure 3.27: Defining and using a Event subclass.

- The operation **foo => bar**, where **foo** and **bar** are UGen's, connects the output of **foo** to the input of **bar**.
- Unit generators are controlled by invoking their member functions over time.
- **gain()** (of type **float**): sets/gets the gain of the UGen's output.
- **last()** (of type **float**): get the last sample computed by the UGen. if UGen has more than one channel, the average of all components channels are returned.
- **channels()** (of type **int**): get the number of channels in the UGen.
- **chan()** (of type **UGen**): return reference to a channel by its integer index (or null if no such channel is available).
- **op()** (of type **int**): specify sample-by-sample operation on at the UGen (the result is used as the actual input into the UGen itself). Values are: 0 : stop - always output 0; 1 : normal operation, sum inputs (UGen's) sample-by-sample (default); 2 : normal operation, subtract inputs starting from the earliest connected; 3 : normal operation, multiply all inputs; 4 : normal operation, divide inputs starting from the earlist connected -1 : passthru - all inputs to the ugen are summed and passed directly to output.
- Three default, global unit generator instances are provided. They are **adc**, **dac**, and **blackhole**.

In addition to chaining UGen's together in a *feedforward* manner, it is also possible to introduce feedback in the network. (Figure 3.28; also see the native Karplus-strong plucked string model example later in this chapter)

UGen's and UAnae may be dynamically connected in this fashion into an audio synthesis/analysis network. It is essential to note that the above only connects the unit generators/analyzers, but does not actually compute audio, unless time is advanced. It is also possible to dynamically disconnect unit generators, using the UnChucK operator (=< or !=>). (Figure 3.29)

```
// connect adc to delay to dac; (feedforward)
adc => Delay delay => dac;

// delay to gain back to itself (feedback)
delay => Gain g => delay;
```

Figure 3.28: Setting up a unit generator network with feedback.

```
// connect SinOsc to dac
SinOsc foo => dac;

// let time pass for 1 second
1::second => now;

// disconnect 'foo' from the 'dac'
foo =< dac;

// let time pass for another second (silence)
1::second => now;

// connect again
foo => dac;

// ...
```

Figure 3.29: Dynamically connecting/disconnecting unit generators.

In ChucK, parameters of unit generators may be controlled and altered at any point in time and at any rate (even sub-sample rate). To set the a value for a parameter of a unit generator a value of the proper type should be ChucKed to the corresponding control function. To read the current value of certain parameters (not all parameters can be read), we may call an overloaded function of the same name. Figure 3.30 demonstrates setting/getting control values. Additionally, one can chain assignments together when assigning a single value to multiple targets.

Note that the parentheses are only needed when the read function is on the very left (i.e., the beginning of a chained chuck statement).

```
// SinOsc to dac
SinOsc foo => dac;
// TriOsc to dac
TriOsc bar => dac;

// set frequency of foo and then bar
500 => foo.freq => bar.freq;

// set one freq to the other
foo.freq() => bar.freq;
// the above is same as:
bar.freq( foo.freq() );
```

Figure 3.30: Asserting/reading control values.

ChucK supports stereo (default) as well as multi-channel audio. **dac** and **adc** are potentially multi-channel UGen's. By default, chucking two UGen's containing the same number of channels (e.g., both stereo or both mono) automatically matches the output channels with the input channels (e.g., left to left, right to right for stereo). Multichannel UGen's mix their output channels when connecting to mono UGen's. Mono UGen's split their output channels when connecting to multi-channel UGen's. Stereo UGen's contain the parameters **left** and **right**, which allow access to the individual channels. It is also possible to address channels by index.

At the time of this writing, ChucK provides around 80 Unit Generators, including oscillators, noise generator, filters, envelopes, delays, as well as a majority of the Synthesis Toolkit (STK) [20]. One idea with ChucK is that due to the flexible timing mechanism, there is a reduced need for writing low-level UGen's or plug-ins in another language (e.g., C++), as such modules can directly specified in ChucK.

3.3.10 Unit Analyzers

Unit Analyzers (**UAna**, singular; *UAnae*, plural) are analyis building blocks, similar in concept to unit generators. They perform analysis functions on audio signals and/or metadata input, and produce analysis results as output. Unit analyzers can be linked to each other as well as to unit generators to form analysis/synthesis networks. Like unit generators, several unit analyzers may run concurrently, each dynamically controlled at different rates. Data passed between UAnae is not necessarily audio samples, and the relationship of UAna computation to time is fundamentally different than that of UGen's (e.g., UAnae might compute on blocks of samples, or on metadata, and on demand). Thus, the connections between UAnae have a different meaning from the connections between UGen's formed with the ChucK operator (=>). This difference is reflected in the choice of a new connection operator, the upChucK operator (=^). Another key difference between UGen's and UAnae is that UAnae perform analysis (only) on demand, via the **upchuck()** function (discussed below). Some additional properties of ChucK unit analyzers are as follows:

- All ChucK unit analyzers extend from unit generators.
- The operation **foo =^ bar**, where **foo** and **bar** are UAnae, connects **foo** to **bar**.
- Unit analyzer parameters and behaviors are controlled by calling / chucking to member functions over time, like unit generators.
- Analysis results are stored in objects called **UAnaBlob**'s. The UAnaBlob contains a time-stamp indicating when it was last computed, and may store an array of floats and/or complex values. Each UAna specifies what information is present in the UAnaBlob it produces.

- All unit analyzers have the function **upchuck()**, which when called issues a cascade of analysis computations for the unit analyzer and any *upstream* unit analyzers on which its analysis depends. In the example of **foo =^ bar**, **bar.upchuck()** will result in **foo** performing its analysis (possibly recursively requesting analysis results from unit analyzers further upstream), then in **bar** using **foo**'s analysis results in its computation. **upchuck()** returns the analysis results inside a UAnaBlob.
- Unit analyzers are specially integrated into the virtual machine such that each unit analyzer performs its analysis **on demand** via **upchuck()**. Combined with timing and concurrency, the programmer has the power to control the analysis process at any point in time and at any desired rate.

3.4 System Design and Implementation

In order to support the features and behaviors of the ChucK language, a variety of system design decisions have been made, and are described in this section. The ChucK system includes a dedicated lexer, parser, typing checker/type system, and virtual machine employing a user-level *shreduler* that *shredules* the *shreds*. We address the salient components of the system, and outline the central shreduling algorithms.

3.4.1 Architecture

ChucK programs are type-checked, emitted into ChucK shreds containing byte-code, and then interpreted in the virtual machine. The shreduler serializes the order of execution between various shreds and the audio engine. Under this model,

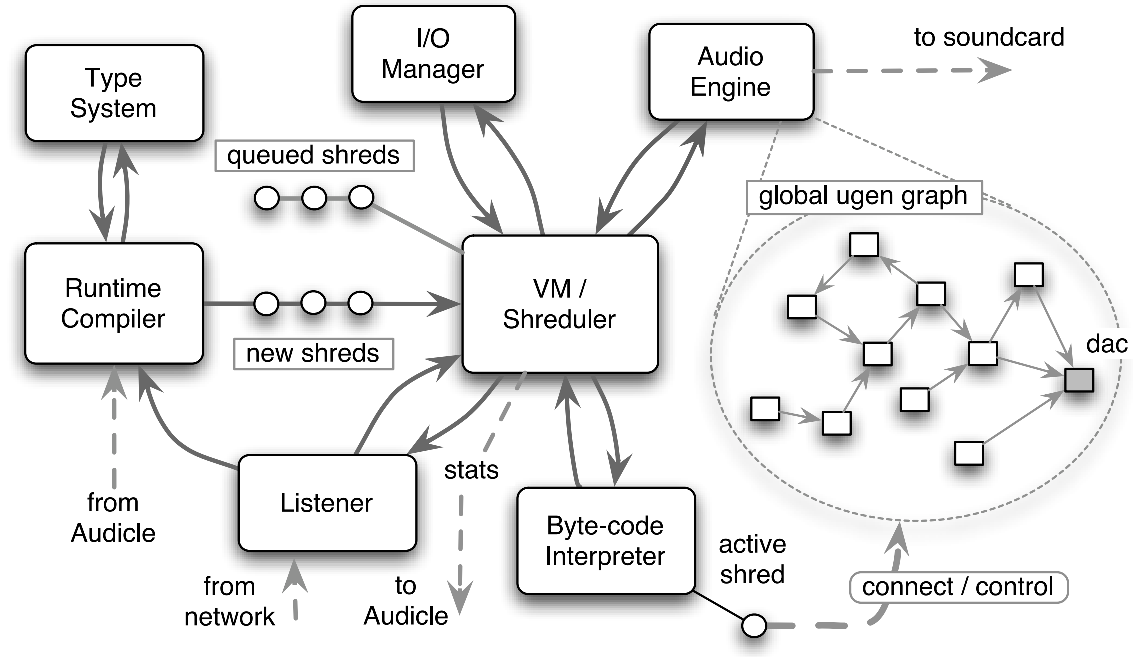


Figure 3.31: ChucK run-time architecture.

shreds can dynamically connect, disconnect, and share unit generators in a global network. Additionally, shreds can perform computations and change the state of any unit generator/analyzers at any point in time. Audio is synthesized from the global unit generator graph a sample at a time by “sucking” samples starting from dedicated UGen “sinks”, such as **dac**. Time as specified in the shreds is mapped by the system to the audio synthesis stream. When a shred advances time, it can be interpreted as the shred shreduling itself to be woken up at some future sample. In this sense, the passage of time is data-driven, and this guarantees that the timing in the shreds is bound to the audio output and not to any other clocks. Furthermore, it guarantees that the final synthesis/analysis result is “correct”, reproducible, and sample-faithful regardless of whether the system is running in real-time or not. Additional processes interface with I/O devices (as necessary) and the runtime compiler. A server listens for incoming network messages. Various parts of the VM can optionally collect real-time statistics to be visualized externally in environments such as the Audicle (see Chapter on Audicle).

3.4.2 Compilation

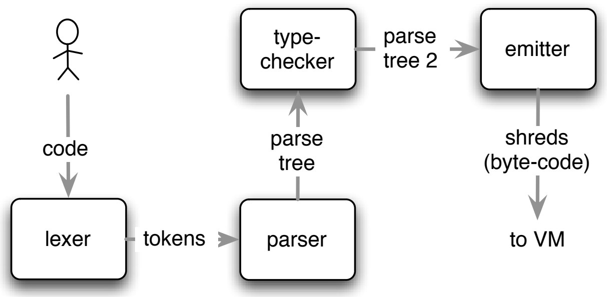


Figure 3.32: Phases in the ChucK compiler.

Compilation of a ChucK program follows the standard phases of lexical analysis, syntax parsing, type checking, and emission into instructions (Figure 3.32). As previously stated, ChucK is imperative and strongly-typed. Programs are emitted into ChucK virtual machine instructions, either as part of a new shred, or as globally available routines. The compiler runs in the same process as the virtual machine, and can compile and run new programs on-demand. By default, all operations, including instruction emission, take place in main memory. This has the advantage of avoiding intermediate steps of writing instructions to disk, and also many costly load-time memory translations which would be necessary if the compiler and VM were to run in separate memory address spaces. The disadvantage of this in real-time is that the compilation must be relatively fast, which precludes the possibility of many advanced compiler optimizations.

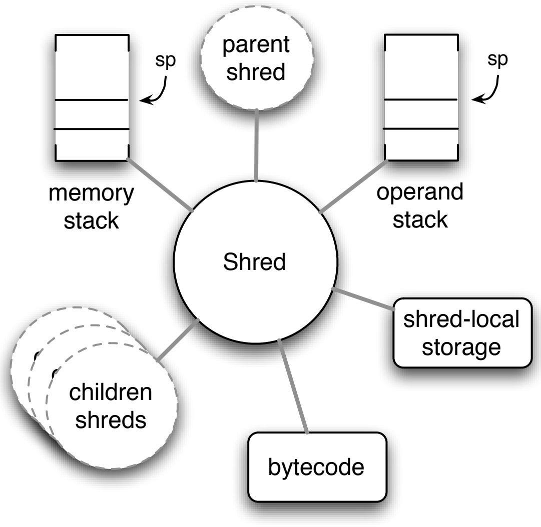


Figure 3.33: A ChucK shred and primary components.

3.4.3 ChucK Virtual Machine + Shreduler

After compilation, a ChucK shred is passed directly to the virtual machine, where it is shreduled to start execution immediately. Each shred has several components (Figure 3.33): (1) bytecode instructions emitted from the source code, (2) an operand stack for local and temporary calculations (functionally similar to hardware registers), (3) a memory stack to store local variables at various scopes, i.e., across function calls, (4) references to children shreds (shreds spawned by the current shred) and a parent shred, if any, and (5) a shred-local view of **now** - which can be a fractional sample away from the system-wide **now**; this is used to maintain sub-sample timing.

The state of a shred is completely characterized by the content of its stacks and their respective pointers. It is therefore possible to suspend a shred between two instructions. Under normal circumstances, however, a shred is suspended only after instructions that advance time. Shreds can spork and remove other shreds.

The shreduler serializes the synchronous execution of shreds with that of the audio engine, while maintaining the system-wide value of the keyword **now**. The unit of **now** is mapped to the number of samples in the final synthesis that have elapsed since the beginning of the program.

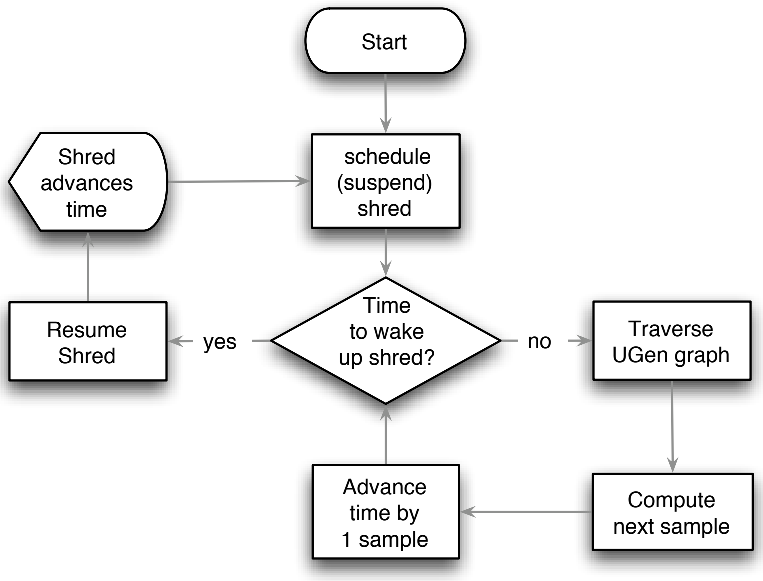


Figure 3.34: Single-shredded shreduling algorithm.

For a single shred, the shreduling algorithm is illustrated in Figure 3.34. A shred is initially shreduled to execute immediately - further shreduling beyond this point is left to the shred. The shreduler checks to see if the shred is shreduled to wake up at or before the current time (**now**). If so, the shred resumes execution in the interpreter until it schedules itself for some future time **T**. At this point, the shred is suspended and the wake-up time is set to **T**. Otherwise, if the shred is not scheduled to wake up at **now**, then the shreduler calls the audio engine, which traverses the global unit generator graph and computes the next sample. The shreduler then advances the value of **now** by the duration of 1 sample, and checks the wake-up time again. It continues to operate in this fashion, interleaving shred execution and audio computation in a completely synchronous manner.

Two points should be noted here. It is possible that a shred misbehaves and never advances time or, in the real-time case, performs enough computation to delay audio. The Halting Problem [87, 78] informs us that the VM cannot hope to detect this reliably. However, it is possible for the user to identify this situation and manually remove a shred from the interpreter. Secondly, the above algorithm is geared towards causal, immediate mode operations in which time can only be advanced towards the future. It is conceivable that this same model can be extended so that shreds can also move backwards in time; this is not discussed in this work.

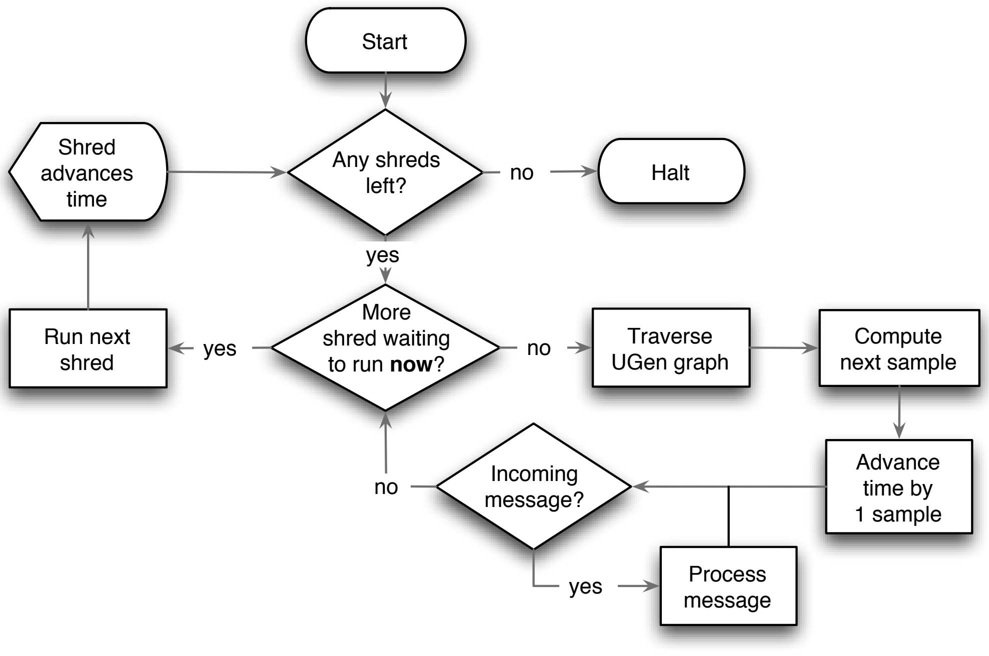


Figure 3.35: Multi-shredded shreduling algorithm, with messaging

For multiple shreds, the mechanism behaves in a similar manner, except the shreduler has a waiting list of shreds, sorted by requested wake-up time. A more comprehensive concurrent shreduling algorithm is shown in Figure 3.35. Before the system-wide **now** is advanced to the next sample, all shreds waiting to run at or before the current time are allowed to execute.

Also, it is possible for a shred to advance time by any amount, even durations less than that of a sample. To support this, each shred keeps track of a shred-local **now**, which is close to the value of the system-wide **now**, but with some fractional sample difference. This allows a shred to shredule itself at practically any increment. This value is compared against the system-wide **now** when determining when to wake up a shred. So it is possible for a shred to run any number of times before the system-wide **now** is advanced to the next sample.

3.4.4 Audio Computation

Unit generators (and more recently, unit analyzers) are created, connected, disconnected, and controlled from code. However, the actual computation of the audio takes place separately in the audio engine. When the shreduler decides that it is appropriate to compute the next sample and advance time, the audio engine is invoked. The global unit generator graph is traversed in depth-first order, starting from one of several well-known sinks, such as **dac**. Each unit generator connected to the **dac** is asked to compute and return the next sample (which may involve first recursively requesting the output of upstream UGen's). The system marks visited nodes so that each unit generator is computed exactly once for every time step. The output value of of each ugen is stored and can be recalled, enabling feedback cycles in the graph. Cyclic recursion is terminated by respecting the marks denoting a node having been visited at the current time, and using the cached output value at previously visited nodes. Additionally, this will result in a single sample delay at some point in the cycle - if the graph remains unchanged, the delay happens in the same manner at every time step.

Another well-known sink is the **blackhole**, which sucks samples like **dac**, but does not play them. **blackhole** is useful for driving standalone unit generators, as well as analysis networks that require no audio output. For instance, one can connect the **dac** to other unit generators, such as FileOut for recording the **dac** to file. These unit generators need to be driven by a sample-rate sink but should not be played. **blackhole** fulfills this purpose.

3.5 Properties

Now that we have described the design goals, the features of the language, as well as its implementation, we now discuss some potentially useful properties of the language.

3.5.1 Time and Programming

The ChucK programmer always codes in *suspended animation*. This property guarantees that time in ChucK does not change unless the programmer explicitly advances it. The value of **now** can remain constant for an arbitrarily long block of code, which has the programmatic benefits of (1) guaranteeing a deterministic timing structure to use and reason about the system and (2) giving a simple and natural mechanism of complete timing control to the programmer. The deterministic nature of timing in ChucK also ensures that the program will flow identically across different executions and machines, free from the underlying hardware timing (processor, memory, bus) and non-deterministic scheduling delays in the operating system kernel scheduler. As a consequence, the programmer is responsible for "keeping up with time" (i.e., specifying when to "step out" of suspended animation and advance time).

Another potentially useful property afforded by the ChucK timing mechanism is that statements that appear in code before the time advancement are guaranteed to evaluate beforehand, and those that appear after the time advancement will evaluate only after the timing or synchronization operation is fulfilled. It is essential to note that blocks of code between operations that advance time are truly *atomic*; statements in each block are considered to take place at the same *logical instant.* This semantic can lead to programs that are significantly easier to specify, debug, and reason about. Furthermore, like the ChucK operator, this approach can further encourage a strong sense of order in the program.

Additionally, the timing mechanism allows feedback loops with single-sample delay, enabling clear representation of signal processing networks, such as the classic Karplus-Strong plucked string physical model [40] (Figure 3.36). Additionally, it is straightforward to implement various extensions of the model [39, 81] as well as a number of other physical models directly in the language – and hear / test them on the spot, making the system an ideal teaching tool for these topics.

3.5.2 Dynamic, Precise Control Rate

The manner with which a shred advances its way through time can be naturally interpreted as the control rate in ChucK. Since the amount of time to advance at each point is determined by the programmer, the control rate can be (1) as rapid (e.g., same or faster than sample rate) and variable (e.g., milliseconds, minutes, days, or even weeks) as the application desires, and (2) dynamically varying with time

```
// feedforward
Noise imp => OneZero lowpass => dac;
// feedback
lowpass => Delay delay => lowpass;

// our radius
.99999 => float R;
// our delay order
500 => float L;
// set delay
L::samp => delay.delay;
// set dissipation factor
Math.pow( R, L ) => delay.gain;
// place zero
-1 => lowpass.zero;

// fire excitation
1 => imp.gain;
// for one delay round trip
L::samp => now;
// cease fire
0 => imp.gain;

// advance time
(Math.log(.0001) / Math.log(R))::samp => now;
```

Figure 3.36: Constructing a classic Karplus and Strong plucked string model.

(since the programmer can compute or lookup the value of each time advancement). Additionally, the power of this dynamic, arbitrary control rate is greatly extended by ChucK's concurrency model, which allows multiple independent control flows to coexist in parallel.

In (Figure 3.37) is a program that moves through time, polling the value of an envelope follower. Note in this code, the programmer has complete control over the poll rate, and can dynamically throttle it at will.

It is possible in ChucK to calculate each sample completely from within the language (though low-level built-in and add-in ChucK modules may be more suitable

```
// patch
adc => Gain g => OnePole p => blackhole;
// square the input
adc => g;
// multiply
3 => g.op;

// set filter pole position
0.99 => p.pole;

// infinite time loop
while( true )
{
    // test
    if( p.last() > 0.01 )
    {
        // detected
        <<< "BANG!!" >>>;
        // wait a bit
        80::ms => now;
    }

    // advance time, also poll rate
    20::ms => now;
}
```

Figure 3.37: An envelope follower (and simple onset detector), based on a leaky integrator. (author: Perry Cook)

for such low-level tasks). All external events, such as MIDI, input devices, and other asynchronous events, are internally handled at a coarser granularity proportional to a tunable latency (e.g., I/O buffer size). Program logic can be specified at any granularity relative to the audio. Thus, the same ChucK timing mechanism can be used to build low-level instruments, as well as high-level compositional elements. The practice of enabling the programmer to operate on an arbitrarily fine granularity is partly derived from the Synthesis Tool Kit (STK) [20], which exposes a manageable programming interface for efficient single sample operations, with additional levels of internal buffering. ChucK builds on this notion to support sample-level computations as well as computations at arbitrarily large intervals, and among concurrent processes.

3.5.3 Synchronous Concurrent Control

Computer audio/music is most often the simultaneity of many parallel sequences of operations, potentially taking place at potentially many distinct rates. Shreds naturally separate each set of independent tasks into concurrent entities running at their own control rates. For example, there might be many different streams of audio samples being generated at multiple control rates; MIDI and OSC messages might arrive periodically (e.g., on the order of milliseconds) from a variety of sources, which control parameters in the synthesis. Concurrently, packets may arrive over the network, while an array of mice and joysticks send control data. At the same time, higher-level musical processes may be computing at yet another concurrent time-scale. In ChucK, it is possible and straightforward to design, specify, and incrementally develop such a system, via shreds and time.

In ChucK, timing and synchronization are duals: explicit timing generates implicit synchronization and explicit synchronization generates implicit timing. Advancing time is an implicit and precise synchronization mechanism for shreds, while explicitly synchronizing on events allows time to advance in the meanwhile.

An extremely important property here is that while shreds are interleaved in time and therefore appear to be concurrent, code executes without preemption; thus every sequence of statements (up to time advance or yields) behaves as a *critical section* or *atomic transaction.* The programmer is not required to designate explicit critical sections with any synchronization in the code.

```
// synthesis patch
Impulse i => TwoZero t => TwoZero t2 => OnePole p;
// formant filters
p => TwoPole f1 => gain g => JCRev r => dac;
p => TwoPole f2 => g;
p => TwoPole f3 => g;

// ... (omitted: initialization code) ...

spork ~ ramp_stuff(); // interpolate pitch and formants
spork ~ do_impulse(); // voice source

while( true ) // main shred
{
    // set next formant targets
    Std.rand2f( 230.0, 660.0 ) => target_f1freq;
    Std.rand2f( 800.0, 2300.0 ) => target_f2freq;
    Std.rand2f( 1700.0, 3000.0 ) => target_f3freq;

    // random walk the scale
    32 + scale[randWalk()] => Std.mtof => freq;
    // set target period
    1.0 / freq  => target_period;

    // wait until next note
    Std.rand2f( 0.2, 0.9 )::second => now;
}

// for shred: generate pitched source, with vibrato
fun void do_impulse()
{
    while( true )
    {
        // fire impulse!
        masterGain => i.next;
        modphase + period => modphase;
        // advance time (modulated to achieve vibrato)
        (period + 0.0001*Math.sin(2*pi*modphase*6.0))::second => now;
    }
}

// for shred: to perform interpolation for various parameters
fun void ramp_stuff()
{
    0.10 => float slew;
    while( true )
    {
        (target_period - period) * slew + period => period;
        (target_f1freq - f1freq) * slew + f1freq => f1freq => f1.freq;
        (target_f2freq - f2freq) * slew + f2freq => f2freq => f2.freq;
        (target_f3freq - f3freq) * slew + f3freq => f3freq => f3.freq;
        0.010 :: second => now;
    }
}
```

Figure 3.38: A concurrent program framework for singing synthesis, naturally balancing source generation, musical parameters, and interpolation in three shreds. (author: Perry Cook)

ChucK imposes no boundaries on the timing structure of a program - it does not make any decision about control rate or timing but instead integrates this decision into the language semantics (which the programmer can easily control). This enables the programmer to create and simultaneously execute any number of shreds - each potentially running at a different control rate. As example, Figure 3.38 shows the code framework for a singing synthesizer where the tasks of musical control (setting vowels, fundamental pitch), source generation (impulse train, modulated for vibrato), and interpolation (smoothly ramp to parameter targets at arbitrary granularity) are represented as three concurrent shreds. This example, while simple, demonstrates the flexibility of shreds and the potential to build and experiment with more complex systems (e.g., complex singing synthesis models such as SPASM [18]).

3.6 Where to go from here

In this chapter, we presented the design, specification, implementation, and properties of the ChucK programming language. Based on these ideas, we next explore two ramifications of ChucK, On-the-fly Programming and the Audicle.

Chapter 4

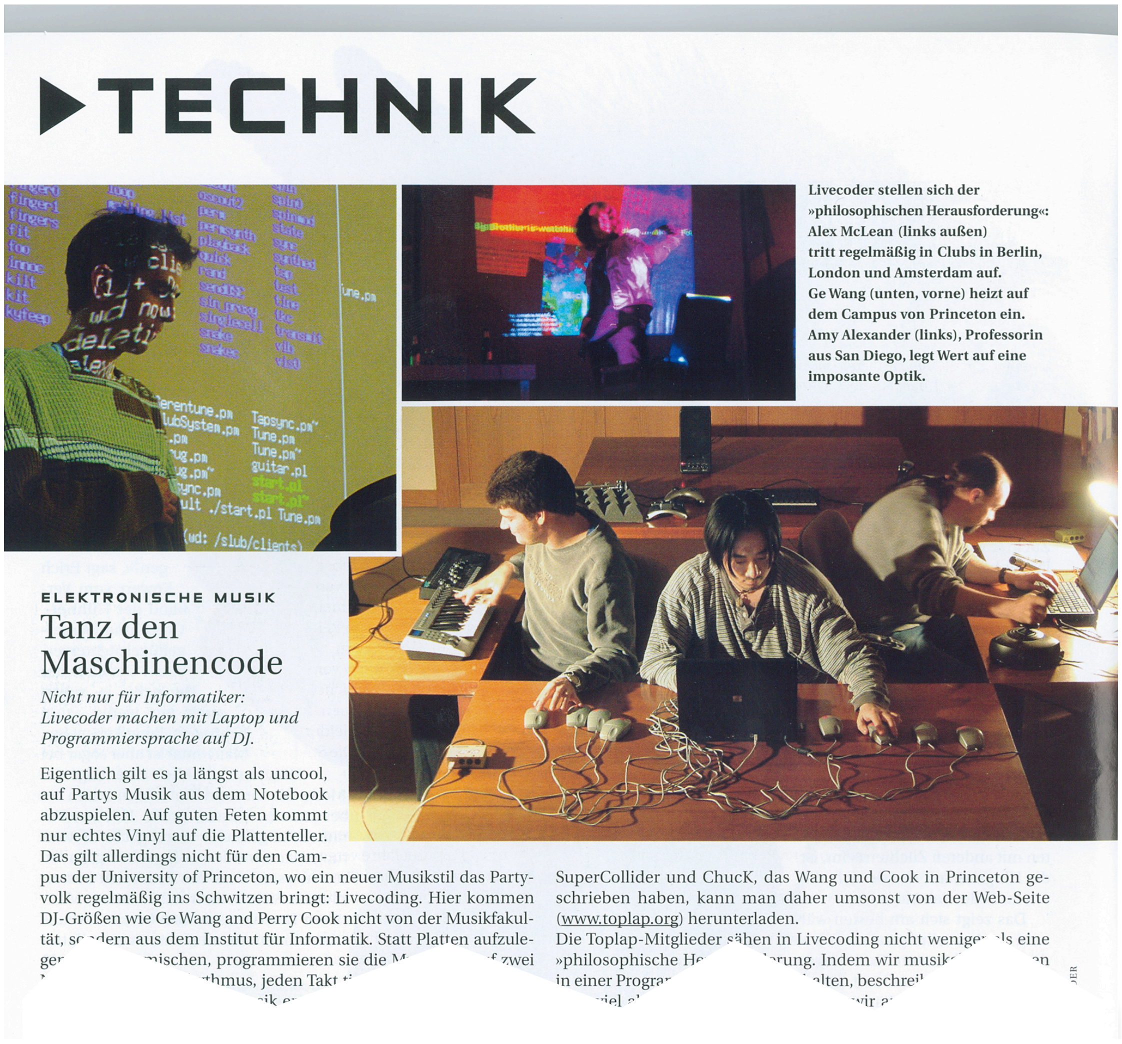
On-the-fly Programming

4.1 Motivation

Due to their fundamental expressive power, programming languages and systems play a pivotal role in the composition, performance, and experimentation of computer audio and electro-acoustic music. Until recently, the design and writing of computer music programs have been limited to off-line development and preparation, leaving only the finished program to "go live". Thus, the gamut of runtime possibilities is prescribed by the functionalities that are programmed ahead of time. By contrast, an *on-the-fly programmable system* provides the ability to write, modify, compile, and execute new/existing code, and to then integrate it into a running program with precise timing and synchronization. The goal of on-the-fly programming (or live coding) is to enable programmers/performers/composers to actively modify the logic and structure of their programs during runtime without having to stop, code, and restart – for the purpose of rapid experimentation, pedagogy, and even live performance. For example, performers could add/change modules in their synthesis or composition programs, or modify mappings to their controllers during a live performance. Similarly, composers can experiment with their programs online, modifying synthesis/analysis components, shaping or perfecting a sound, or changing compositional elements, while being able to hear the result immediately.

Performers have used runtime programmable elements during live performance and rehearsal. Examples go back as far as to Jim Horton, Tim Perkis, and John Bischoff of The League of Automatic Composers, who tweaked live electronics with microcomputers (KIMs) during performance, George Lewis in creating *Voyager* [51], the network group The Hub, who used languages like FORTH to modify their systems online, to more recent laptop computer musicians who compose and perform via various on-the-fly tools, including command-line, shell scripts, and homemade software systems [17]. The latter include Alex McLeans's feedback.pl, Dave Griffith's Fluxus, JITLIB for SuperCollider, libraries/systems for Python, Ruby, and others [105] (Figure 4.1), as well as many others that can be found as part of TOPLAP [82]. As mentioned in Chapter 2 (A History of Music and Programming), the barriers to entry (social, economical, as well as psychological) have been drastically reduced in recent years, perhaps a sign that computers have become truly pervasive to the point where they act as nature extensions of our everyday lives. In the context of music, the computer is a platform for creation and live performance - no longer just as an end, but also as a self-definable means to achieving musical results.

The features of the programming tool inevitably shape both the approaches by which tasks are implemented as well as the end product. By bringing the power and expressiveness of the programming language into the runtime setting, an on-the-fly programming system has the potential to fundamentally enhance the real-



▶TECHNIK

Livecoder stellen sich der »philosophischen Herausforderung«: Alex McLean (links außen) tritt regelmäßig in Clubs in Berlin, London und Amsterdam auf. Ge Wang (unten, vorne) heizt auf dem Campus von Princeton ein. Amy Alexander (links), Professorin aus San Diego, legt Wert auf eine imposante Optik.

ELEKTRONISCHE MUSIK

Tanz den Maschinencode

Nicht nur für Informatiker: Livecoder machen mit Laptop und Programmiersprache auf DJ.

Eigentlich gilt es ja längst als uncool, auf Partys Musik aus dem Notebook abzuspielen. Auf guten Feten kommt nur echtes Vinyl auf die Plattenteller. Das gilt allerdings nicht für den Campus der University of Princeton, wo ein neuer Musikstil das Partyvolk regelmäßig ins Schwitzen bringt: Livecoding. Hier kommen DJ-Größen wie Ge Wang and Perry Cook nicht von der Musikfakultät, sc dern aus dem Institut für Informatik. Statt Platten aufzulegen mischen, programmieren sie die M f zwei thmus, jeden Takt ik e

SuperCollider und ChucK, das Wang und Cook in Princeton geschrieben haben, kann man daher umsonst von der Web-Seite (www.toplap.org) herunterladen.
Die Toplap-Mitglieder sähen in Livecoding nicht weniger ls eine »philosophische He rung. Indem wir musik en in einer Progra lten, beschrei viel a wir a

Figure 4.1: An article about live coding, published in Zeitwissen in 2006.

time interaction between the performer/composer and the systems they create and control. Code becomes a real-time, expressive instrument [97]. We believe that such a potential is worth exploring. In this section, we define on-the-fly programming and provide a formal programming model based on ChucK, leveraging its properties of timing and concurrency, as well as the ChucK virtual machine. In addition, we discuss an open on-the-fly programming aesthetic.

4.2 Challenges

In order to bring the power and general expressiveness of programming languages into an on-the-fly programming setting, we have identified several fundamental challenges that must be addressed:

- **Modularity** – code sections should be modular so the programmer can reason about them or modify them independently. Furthermore, the augmented code must work together in the same address space and namespace.
- **Timing** – there must be a strong consistency and notion of time between the existing and new parts of the program. On-the-fly code segments need to start and stop with precision.
- **Conciseness, expressiveness, and manageability** – given the substantial time constraints of live coding, we ask: how can ideas be realized concisely and *expressively* in code? How do we reason about time and data flow easily?
- **Flexibility** – how flexible is the system? Does it allow programmers to take advantage of the expressive power of programming languages in a real-time setting?

4.3 A ChucKian Approach

In this section, we describe the ChucKian on-the-fly programming model. We do so in terms of external and internal semantic. We reason about properties in the model. We show that just as concurrency in ChucK is a natural extension of the timing mechanism, we can leverage the timing mechanism and concurrency to address the challenges of on-the-fly programming.

4.3.1 External Interface

The on-the-fly programming model, at a high-level, can be described in the following way. A ChucK virtual machine begins execution, computing audio (as necessary), keeping time, and waiting for incoming shreds. A ChucK shred can be on-the-fly assimilated into the virtual machine, sharing the program address space and global timing mechanism, and is said to be *active*. Similarly, an active shred can be dissimilated, or removed from the virtual machine, or it can be suspended or be replaced by another shred. This interface is designed to be simple, and delegates the actual timing and synchronization logic to the code within the shred, leaving this flexibility (and responsibility) to the programmer.

The high level commands to the external interface are listed below. They can be invoked on the command line, in ChucK programs (as functions calls to the machine and compiler objects), over the network, via customized graphical interfaces, or by other appropriate means.

- **Execute** – begins a new instance of the virtual machine in a new address space. Typically, this operation is used at the beginning of the session. Multiple instances of the virtual machine can coexist. The shreduler begins to keep track of time.
- **Add** – type-checks and compiles a new shred (from a ChucK source file, a string containing ChucK code, or a pre-compiled shred). If there are no compilation errors, the shred is allocated and sporked in the virtual machine with an unique ID. A new virtual stack is allocated, and the shred is shreduled immediately to execute from the beginning. When **add** fails due to compilation errors, the virtual machine continues to run as before while the programmer can attempt to debug, correct, and re-add the code.

- **Remove** – removes a shred by ID or name from the virtual machine. The shred and its child objects are finalized and reclaimed.
- **Replace** – invokes a **remove** operation followed by an **add** – the two components happen *instantaneously* in ChucK time.
- **Status** – queries the status of the virtual machine for the following types of information: (1) a list of active/suspended shred ID's, source/filename, duration since assimilation (spork time), and (2) information on virtual machine state: currently executing shred, shreduler timeline, and other statistics by various parts of the system.

For example, Figures 4.2 and 4.3 show code that adds, replaces, and removes two shreds using separate methods.

```
# start VM with "infinite time-loop"
shell%> + `while(true) 1::second => now;`
# add foo.ck
shell%> + foo.ck
# replace shred 0 with bar.ck
shell%> = 0 bar.ck
# remove all shreds
shell%> --remove.all
```

Figure 4.2: "external" ChucK shell commands for adding/replacing code.

```
// add shred from file "foo.ck"
Machine.add( "foo.ck" ) @=> Shred @ foo;
// advance time by 500 milliseconds
500::ms => now;
// replace foo with "bar.ck"
Machine.replace( foo, "bar.ck" ) @=> Shred @ bar;
// advance time by 2 seconds
2::second => now;
// remove bar
Machine.remove( bar );
```

Figure 4.3: "internal" ChucK shell commands for adding/replacing code.

The *code-runs-code* feature is powerful because it allows a program to self-manage shreds on-the-fly with sample-synchronous precision. Users can also assimilate shreds that themselves add (potentially many) additional shreds, each with precise timing. Because the compiler and the virtual machine run in the same process, much of the intermediate processing can be eliminated. Finally, the ability to evaluate strings as code at runtime opens the possibility for self-generating on-the-fly programs with fast compilation-to-runtime response. The status feedback is helpful for quickly surveying the state of the system and is particularly useful in an on-the-fly setting because it can identify hanging or non-cooperative shreds. For example, if the system runs a shred containing an infinite loop that fails to advance time, it will cause the virtual machine to hang indefinitely. However, the on-the-fly programmer can identify and remove misbehaving shreds from the virtual machine manually, resulting in reduced interruption to the performance or session. While this recovery mechanism is far from perfect, it can be much more advantageous than killing the system and restarting. Additionally, it can help the composer/performer tweak the system by identifying shreds that are taking too much CPU time and optimize them individually. This high-level framework uses concurrent shreds as modules and provides a means of managing them. This has led to new interfaces for expressive audio coding, in the Audicle and miniAudicle (discussed in Chapter 5, Audicle). (In performance situations, sheer "redundancy" can also help, for example in the form of coding with a partner, or even simply running several instances of ChucK on multi-core machines!)

4.3.2 Internal Semantics

The approach described in this subsection deals with the problem of precise timing between on-the-fly modules. The goal is to provide a consistent and accurate mechanism for shreds to synchronize with each other. In our model, the semantics are natural extensions of the ChucK timing mechanism. By querying and manipulating time using the special variable **now**, the programmer can determine the current time, and specify how the code should respond. By the properties of ChucK timing and concurrency: (1) **now** always holds the current ChucK time, (2) changing the value of **now** advances time in ChucK and has the side effect of blocking the current shred (allowing audio and other shreds to compute) until **now** holds the value that was assigned to it, (3) if **t** is of type **time**, **t** => **now** advances time until **t** equals **now**, (4) if **d** is of type **dur** (a duration), **d** +=> **now** advances time by **d**. We illustrate this below with some common code segments that synchronize to time (Figures 4.4 to 4.7).

```
// let time pass
now + 10::second => time later;
later => now;

// (or alternately)
10::second +=> now;
```

Figure 4.4: Two methods to "synch" with a later time.

```
// synch/advance to time t
t => now;
```

Figure 4.5: "Synching" to some absolute time.

The simple timing building blocks allow programmers to precisely specify many more timing and synchronization behaviors. These statements can be placed and

```
// period to synchronize to
120::ms => dur T;
// advance time by remainder
T - (now % T) +=> now;
```

Figure 4.6: Define a period; synchronize to next period boundary.

```
// advance time by remainder, plus offset
T - (now % T) + D +=> now;
```

Figure 4.7: Synchronize to period boundary, plus offset.

combined at arbitrary points in the code. For example, to initialize time-based synchronizations in a piece of code, directives may be placed near the beginning of a shred to synchronize to time before moving on.

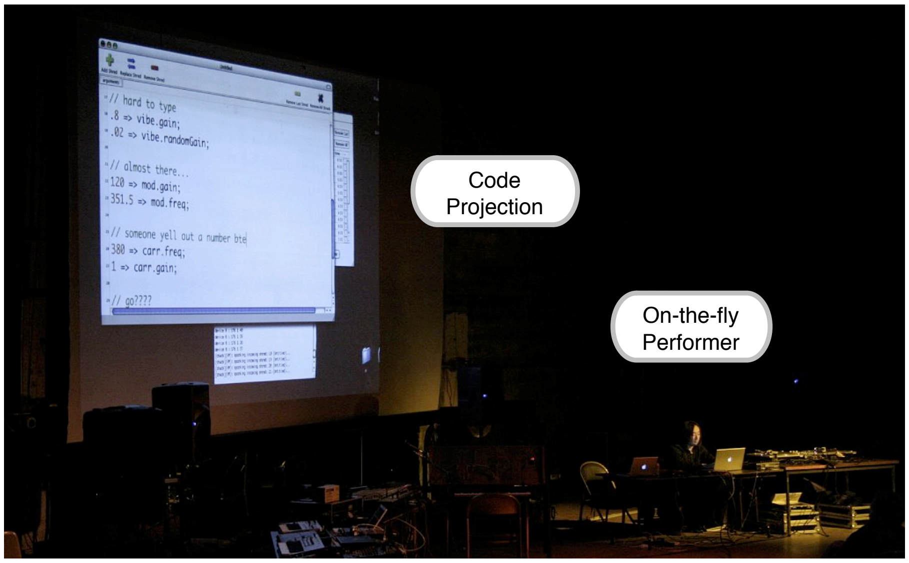
4.4 An On-the-fly Aesthetic

Our on-the-fly aesthetic is one where the process of programming (both action and thought) is conveyed to the observer/listener. In the classroom, on-the-fly programming can be a powerful vehicle for showing how to construct a particular algorithm step by step, while maintaining both a visual and sonic (and hopefully also mental) footprint throughout the entire process. Students can immediately hear how modifying parts of an algorithm can affect the result. In terms of performance, it addresses two important issues in computer music performance. First, it can be argued that many technical and aesthetic intentions are often difficult to discern in performance. The on-the-fly programming aesthetic helps address this concern (when desired), for it provides a way for the audience to perceive both the intention and the result. A single performer configuration can be seen in Figures 4.8 and 4.9. A two-performer schematic and realization can be seen in Figures 4.10 and

4.11. In the experience of the author, the two person configuration is much more enjoyable and fun, for it not only alleviates the stress of coding under pressure by naturally load balancing between the players (as one performer "riffs", the other might begin coding the next section), but also provides much more opportunity for musical interplay (especially since the instrument itself is, by definition, adaptable). In the extreme case, live coding of this kind has been carried out on an orchestral scale, with 15 to 20 live coders contributing to a single sonic and musical entity (this is discussed in Chapter 6, Applications).

The second problem that the on-the-fly aesthetic addresses is the issue of virtuosity in computer music. On-the-fly programming provides a platform where the performer is able to render various types of mastery and creativity that can be immediately appreciated, or at least perceived. While typing speed alone may or may not inspire, the general expressive power of programming languages opens unlimited possibilities for clever approaches and beautiful design. The timing semantics make ChucK code straightforward to follow, allowing the audience to more quickly and easily appreciate the design and construction of on-the-fly programs.

While this framework has many desirable properties, it is still unpolished and unwieldy in many respects, partly because writing (and thinking about) code inherently takes time. Future work may look into programming tools that better understand the deep structure of the program being written and facilitates writing and debugging on-the-fly. Also, it would be interesting to investigate reducing modular granularity, allowing finer sections of code to be runtime modified. In the next chapter, we present a graphical on-the-fly environment designed to both facilitate and visualize the on-the-fly programming process.



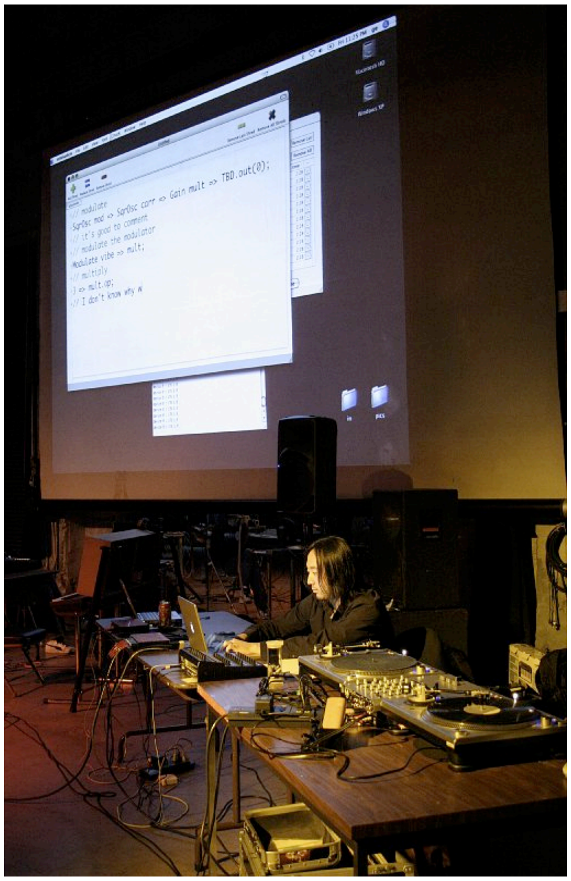
```
// hard to type
.8 => vibe.gain;
.02 => vibe.randomGain;

// almost there...
120 => mod.gain;
351.5 => mod.freq;

// someone yell out a number bte
380 => carr.freq;
1 => carr.gain;

// go????
```

Figure 4.8: An on-the-fly programmer/performer and code projection.



```
// modulate
SqrOsc mod => SqrOsc carr => Gain mult => TBD.out(0);
// it's good to comment
// modulate the modulator
Modulate vibe => mult;
// multiply
3 => mult.op;
// I don't know why w
```

Figure 4.9: An on-the-fly programmer/performer and code projection (close-up).

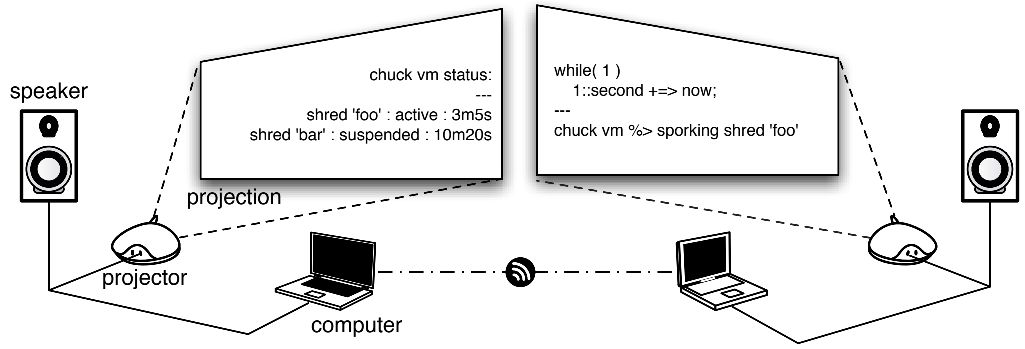

speaker
chuck vm status:
---
shred 'foo' : active : 3m5s
shred 'bar' : suspended : 10m20s
while(1)
1::second +=> now;
---
chuck vm %> sporking shred 'foo'
projection
projector
computer

Figure 4.10: A schematic for a double-projection, on-the-fly duet.

Figure 4.11: A On-the-fly Programming collage, prepared for Art Gallery performance at SIGGRAPH 2006.

Chapter 5

The Audicle

5.1 Introduction

Software environments play a pivotal role in the creation and performance of computer music, not only in terms of providing means of working with sound, but also in encouraging ways of *thinking* about ideas might be realized. *Development environments* provide the setting to design/implement audio and music algorithms, whereas *runtime environments* realize and render these algorithms into sound (and images), and allow performers to interact with the system. In this chapter, we present a new type of audio programming environment that integrates the programmability of the development environment with the immediate feedback of the runtime environment. The result, called the *Audicle*, is an integration of a "smart" editor, compiler, virtual machine, and debugger – all running in the same address space, sharing data, and working together at runtime. We believe these types of augmentation have the potential to fundamentally enhance the way we write, visualize, and interact with audio programs. This chapter discusses the main components of the

Audicle, and show that it not only provides a useful class of programming tools for real-time composition and performances, but also motivates a new type of on-the-fly programming aesthetic – one of visualizing the audio programming process.

5.1.1 Motivation

A simple but important question to ask here is: *why* investigate the programming environment? We believe that the programming language and environment fundamentally influence how we think about and write programs. ChucK, as a programming language, provided different ways of reasoning about time, data-flow, and concurrency. The Audicle is designed to enhance and complement these features, to make some of them more accessible and perhaps more *enjoyable* to use for sound and music.

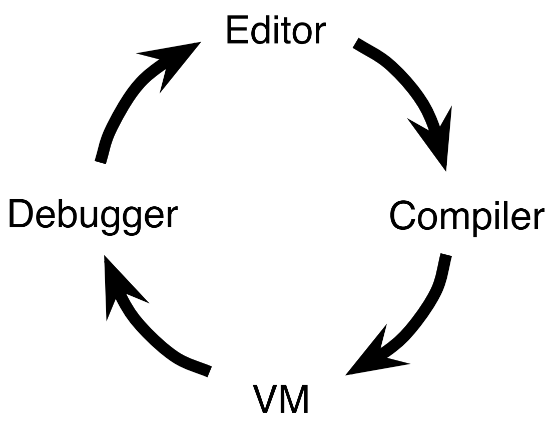


Figure 5.1: Completing the loop. The Audicle strives to bridge runtime interactions with development-time elements.

The Audicle differs from traditional environments in the following ways. Conceptually, it brings the editor and compiler into the runtime environment, in an effort to support a greater level of interactivity in the programming process (Figure 5.1). Secondly, it is tightly coupled with a programming language – in this case, ChucK. This coupling leverages and enhances the ChucK properties of precise timing and concurrency. This is different from systems like Max and Pure Data, where the environment essentially *is* the language. The Audicle aims to complement the language and to enhance the ability to rapidly develop and visualize programs both offline and on-the-fly. Thirdly, the Audicle embodies the aesthetic and mentality of visualizing the programming process and the state of the runtime system. The various goals and considerations in the design are as follows.

- **Context-sensitivity**. The environment should allow code to be clearly entered and represented. Also, it should have some knowledge of the structure and revision history of the program as well as runtime information (such as program statistics) – and use this information to aid the programmer to more easily write code.

- **On-the-fly Programming**. On-the-fly programming is the practice of coding at runtime – while the program is running. The Audicle aims to complete the development/runtime loop by bringing the editor and compiler to the virtual machine, and vice versa. By making components accessible to each other, new interfaces and paradigms for runtime audio programming may avail themselves to the programmer.

- **Different Views**. Having different views of the same program can be useful to writing and fine-tuning code. The Audicle should allow a program to be viewed and manipulated in many ways: as concurrent code, syntactic/semantic representations, or in terms of timing and synchronizations. Additionally, the Audicle is a visualization of the *process* of on-the-fly programming; this has potential to a useful performance and educational tool.

- **Minimalist Design**. The Audicle provides a minimal interface, and relies on the underlying interactions of the language and the multiple viewing models to achieve a great deal of expressiveness and power, while trying not to impose any particular programming style.

5.1.2 Related Environments

An *environment*, in the context of this investigation, is defined as a comprehensive software setting in which programming and/or runtime control is carried out or facilitated. Many environments have been developed for programming, performance, and composition. We examine some, as well as several related environments not specifically intended for audio and music.

Development environments provide a setting to write and edit programs, and often include a compiler and debugger. Examples include graphical environments such as Max/MSP [68] and Pure Data (Pd) [69], integrated development environments (IDEs) for text-based languages such as Java and C/C++, Nyquist [24], and SuperCollider [58], and software frameworks such as Ptolemy [49]. These environments allow code, flow graphs, and other programming constructs to be entered, compiled, and run.

Runtime environments, on the other hand, provide an engine and a related set of interface elements for manipulating parameters at runtime (and often in real-time). Examples include the performance mode of Max/MSP, as well as Real-Time CSOUND [91], and Aura [25]. These environments compute audio in real-time, taking in data from input devices and UI elements, and may also display graphical or video feedback.

On-the-fly environments possess elements of both programming and runtime systems, offering the capability to modify the structure and logic of the executing program itself. Several existing environments possess varying degrees of on-the-fly capabilities. Max and Pd give programmers ability to change aspects of their patches at runtime. The SuperCollider environment allows for synthesis patches to be sent and added to a server in real-time, as well as engage in live coding via libraries like JITLIB [17]. Another interesting system for runtime graphical and virtual-reality programming is Alice [66], which allows users to create a virtual world, and to add and modify behaviors on-the-fly using a high-level scripting language (Python in this case). This rapid-prototyping graphical environment is notable for having no hard distinction between development and runtime. Similarly, MATLAB [56], while not intended as a real-time programming tool, has a command line that directly uses statements from the language and embodies a similar *immediately-run* aesthetic.

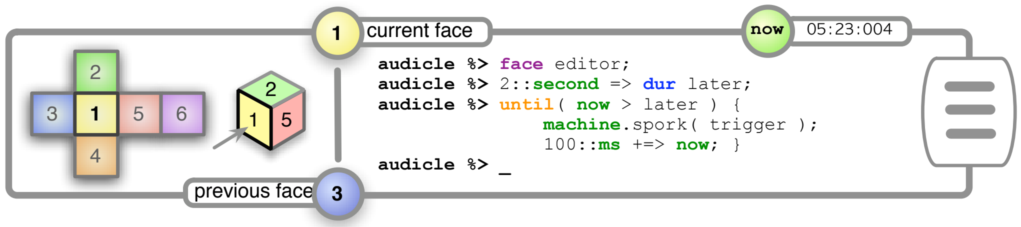
5.2 Audicle Design

The Audicle is a graphical, on-the-fly audio programming environment based on the semantics of ChucK's strongly-timed programming model and on-the-fly programming. These features are strengthened *visualization* in the Audicle. Thus, the graphical nature of the Audicle is given much consideration in the design; it is to be visually meaningful, and open to customization. The idea is to provide a set of tools and visualizations that can be combined into more complex configurations and usages. Much of the information is conveyed by 3D shapes, which can be viewed from virtually any viewpoint or distance, and rendered exclusively using 3D graphics.

The Audicle makes no distinction between development and runtime: all components are fully accessible at runtime. This integration is based on the ChucK compiler and virtual machine – augmented with a smart editor and interfaces for viewing concurrency, timing, and system state. As in ChucK, data-flow and time are fundamentally decoupled. Also, the Audicle's architecture adopts a decoupled simulation model for virtual reality [77]. In this model, the simulation can operate at an arbitrary rate independent of the graphics rendering-rate, leading to smoother graphics and more flexibility in the simulation algorithms. In the Audicle, audio synthesis, graphics, and simulation are loosely-coupled, with the highest priority given to audio computations and the virtual machine.

5.3 Faces of the Audicle

Out of the desire to provide a simple, "graspable" virtual environment and interface, the various facets of the Audicle are mapped and displayed on the faces of a virtual cube, called the *Audicube* (though it's still unclear whether this was a good idea). This 3D object with different faces can be seen as Audicle's way of representing related material from different perspectives. At any time, the user can interact with one face, and has the ability to move to other faces by using hotkeys, graphical interface, and Audicle shell commands. There is a command-line console (Figure 5.2) that can be invoked to appear over the currently active face, allowing on-the-fly programming and other commands to be executed.



```
audicle %> face editor;
audicle %> 2::second => dur later;
audicle %> until( now > later ) {
                machine.spork( trigger );
                100::ms +=> now; }
audicle %> _
```

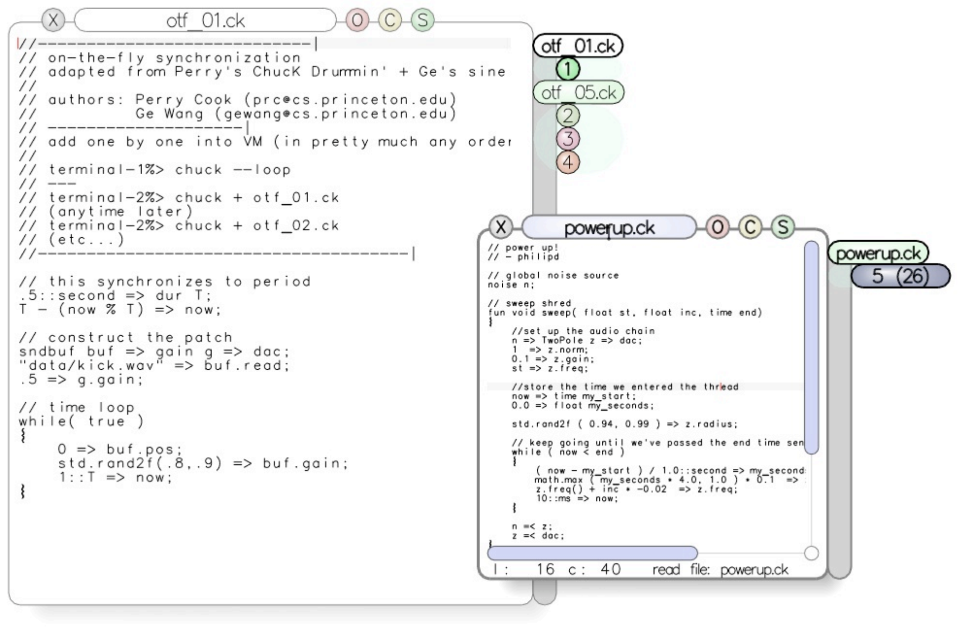
Figure 5.2: The Audicle Console. The cube interface (left) can be used to graphically navigate the AudiCube. The command line prompt on the right accept text commands.

5.3.1 The ShrEditor

The first face of the Audicle, the ShrEditor, is a place for the programmer to write, organize, and listen to ChucK programs (Figure 5.3). One can open and modify existing ChucK programs or create new ones. Code can be *sporked* by clicking the green "S" circle at the top of the buffer. The code runs immediately, allowing the result to be heard. Error messages are displayed in pop-up boxes.

Once a shred is sporked, a numbered circle will appear on the right side of the buffer window. This is a visual representation of a running version of the shred (the number is the shred's ID). Different versions appear under revision tabs, keeping a history that the programmer can browse and recall (Figure 5.4). When a program is modified, the ShrEditor will automatically track it. The programmer can also drag a version to split it from from the initial buffer (Figure 5.5), and view it in its own window.

When many buffer windows are open, it may be difficult to locate a particular buffer of code (Figure 5.6). To facilitate this, the ShrEditor allows users to "drag and throw" buffers, and literally *forage* through a cluttered environment (this doesn't necessarily help the clutter, but can help find a piece of code with less mouse movements, plus the interaction is potentially cool to look at, especially for live coding audiences). Additionally, the "throwing" of windows can be can seen as a "physical" analogy to "chucking".



```
//--------------------------|
// on-the-fly synchronization
// adapted from Perry's ChucK Drummin' + Ge's sine
//
// authors: Perry Cook (prc@cs.princeton.edu)
//          Ge Wang (gewang@cs.princeton.edu)
// --------------------|
// add one by one into VM (in pretty much any order
//
// terminal-1%> chuck --loop
// ---
// terminal-2%> chuck + otf_01.ck
// (anytime later)
// terminal-2%> chuck + otf_02.ck
// (etc...)
//--------------------------------------|

// this synchronizes to period
.5::second => dur T;
T - (now % T) => now;

// construct the patch
sndbuf buf => gain g => dac;
"data/kick.wav" => buf.read;
.5 => g.gain;

// time loop
while( true )
{
    0 => buf.pos;
    std.rand2f(.8,.9) => buf.gain;
    1::T => now;
}
```

```
// power up!
// - philipd

// global noise source
noise n;

// sweep shred
fun void sweep( float st, float inc, time end)
{
    //set up the audio chain
    n => TwoPole z => dac;
    1  => z.norm;
    0.1 => z.gain;
    st => z.freq;

    //store the time we entered the thread
    now => time my_start;
    0.0 => float my_seconds;

    std.rand2f ( 0.94, 0.99 ) => z.radius;

    // keep going until we've passed the end time sen
    while ( now < end )
    {
        ( now - my_start ) / 1.0::second => my_second
        math.max ( my_seconds * 4.0, 1.0 ) * 0.1  =>
        z.freq() + inc * -0.02  => z.freq;
        10::ms => now;
    }

    n =< z;
    z =< dac;
}
```

```
l :   16  c :  40     read  file:  powerup.ck
```

Figure 5.3: The ShrEditor: a version-tracking on-the-fly editing interface.

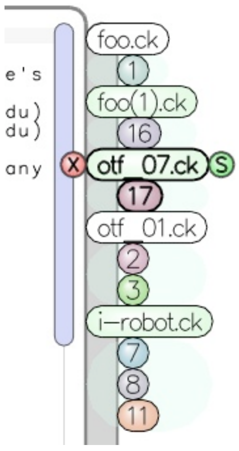
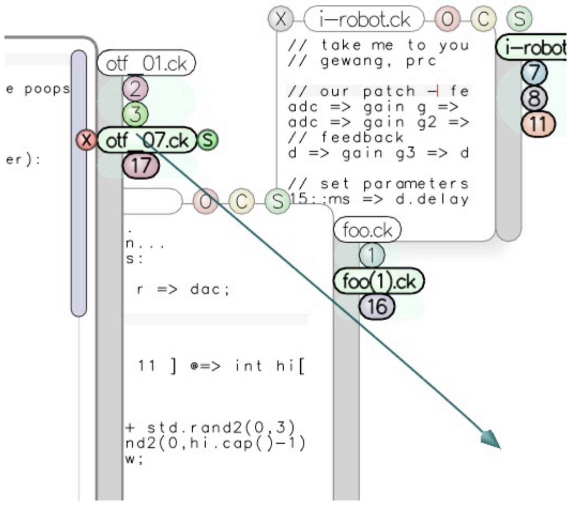


Figure 5.4: "Grapes" represent running shreds, grouped by revision.

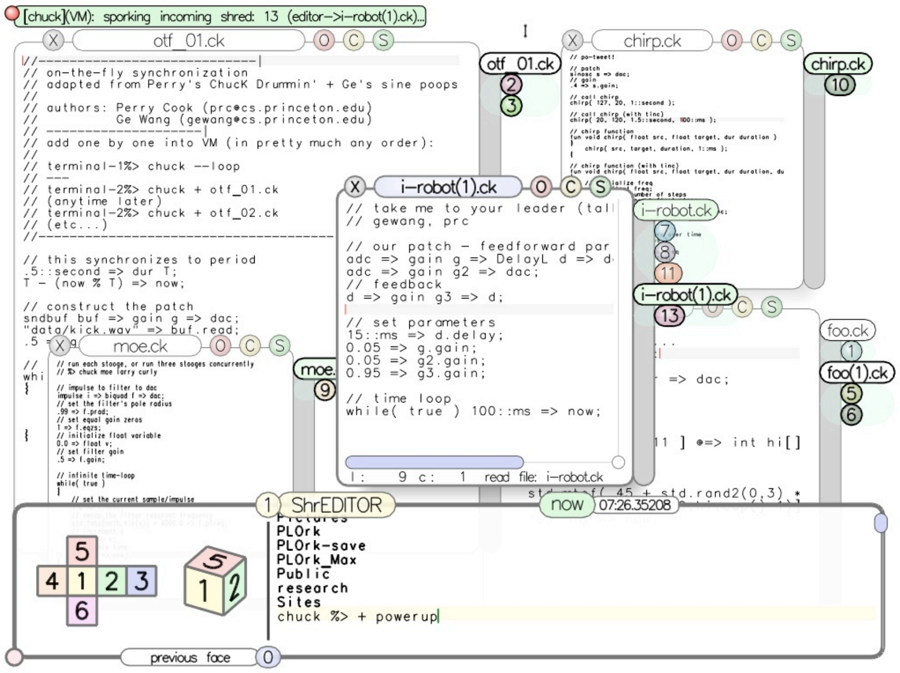


```
// take me to you
// gewang, prc

// our patch ⊣ fe
adc => gain g =>
adc => gain g2 =>
// feedback
d => gain g3 => d

// set parameters
15::ms => d.delay
```

Figure 5.5: One can drag revisions to split text buffers.



```
[chuck](VM): sporking incoming shred: 13 (editor->i-robot(1).ck)...
```

```
//----------------------------|
// on-the-fly synchronization
// adapted from Perry's ChucK Drummin' + Ge's sine poops
//
// authors: Perry Cook (prc@cs.princeton.edu)
//          Ge Wang (gewang@cs.princeton.edu)
// --------------------|
// add one by one into VM (in pretty much any order):
//
// terminal-1%> chuck --loop
// ---
// terminal-2%> chuck + otf_01.ck
// (anytime later)
// terminal-2%> chuck + otf_02.ck
// (etc...)
//--------------------------------------------

// this synchronizes to period
.5::second => dur T;
T - (now % T) => now;

// construct the patch
sndbuf buf => gain g => dac;
"data/kick.wav" => buf.read;
```

```
// run each stooge, or run three stooges concurrently
// %> chuck moe larry curly

// impulse to filter to dac
impulse i => biquad f => dac;
// set the filter's pole radius
.99 => f.prad;
// set equal gain zeros
1 => f.eqzs;
// initialize float variable
0.0 => float v;
// set filter gain
.5 => f.gain;

// infinite time-loop
while( true )
{
    // set the current sample/impulse
```

```
// take me to your leader (tall
// gewang, prc

// our patch - feedforward par
adc => gain g => DelayL d => d
adc => gain g2 => dac;
// feedback
d => gain g3 => d;

// set parameters
15::ms => d.delay;
0.05 => g.gain;
0.05 => g2.gain;
0.95 => g3.gain;

// time loop
while( true ) 100::ms => now;

l:    9 c:    1   read file: i-robot.ck
```

```
// po-tweet!

// patch
sinosc s => dac;
// gain
.4 => s.gain;

// call chirp
chirp( 127, 20, 1::second );

// call chirp (with tinc)
chirp( 20, 120, 1.5::second, 100::ms );

// chirp function
fun void chirp( float src, float target, dur duration )
{
    chirp( src, target, duration, 1::ms );
}

// chirp function (with tinc)
fun void chirp( float src, float target, dur duration, du
```

```
Pictures
PLOrk
PLOrk-save
PLOrk_Max
Public
research
Sites
chuck %> + powerup
```

Figure 5.6: Many on-the-fly coding buffers.

5.3.2 VM-Space

The VM-Space is a useful face for quickly viewing the audio synthesized by ChucK programs (Figure 5.7). The visualization is based on the **sndpeek** visualization software [61]. Stereo/multichannel audio is mixed to mono before being visualized. The blue line on top renders the time-domain waveform. The green lines below make up a waterfall plot of the short-time fourier transform (STFT). This information is useful for debugging sound synthesis algorithms, allowing the programmer to observe the sound, coupled with two of its visual representation.

By clicking on the yellow sphere to the right side of the window, one can change the type of the transform window (e.g., hann, hamming, blackmann-harris, or rectangular). Clicking on the red sphere displays the current window. By clicking and dragging anywhere within the Audicle window, one can view the STFT waterfall plot from different angles.

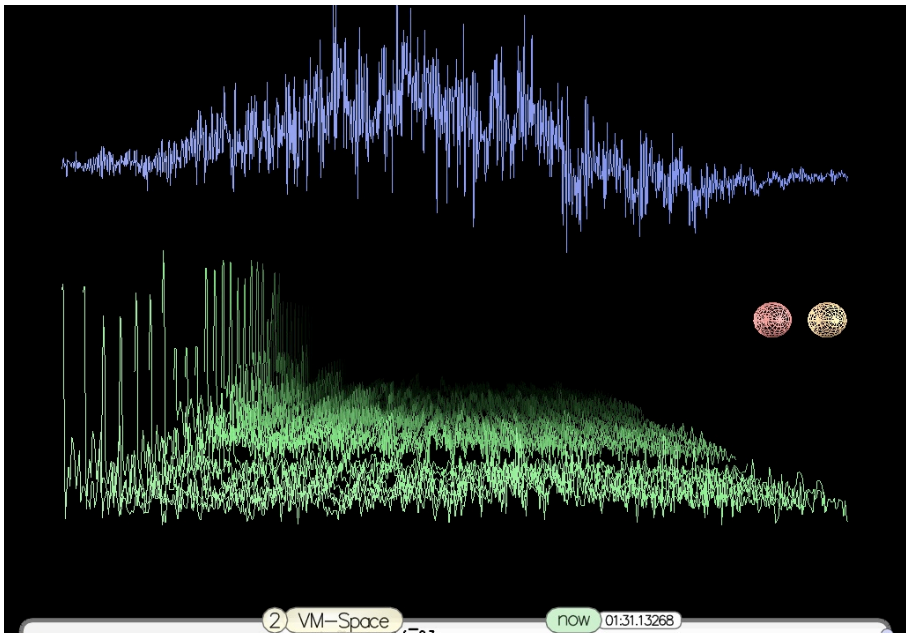


Figure 5.7: VMSpace: Audicle face to visualize real-time audio and spectra.

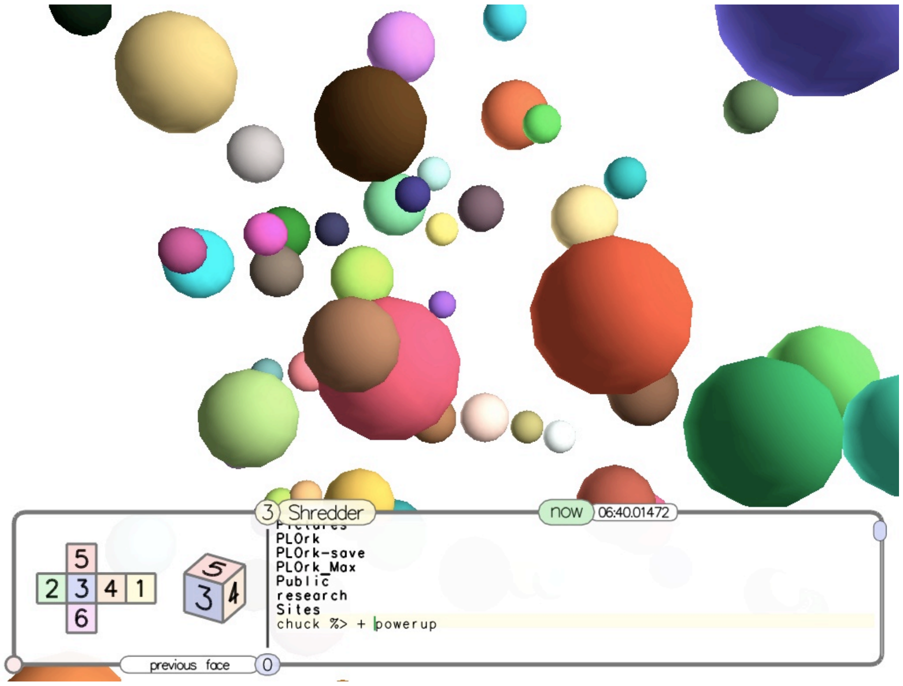
5.3.3 Shredder

The third face of the Audicle, the Shredder, gives a visual representation of the shreds currently running in the ChucK Virtual Machine, as well as a summary of statistics about shreds, updated in real-time.

Each shred is visualized as a separate colored sphere. The spheres rotate on their own, at an angular velocity proportional to the number of virtual machine instructions executed per second. By clicking and dragging anywhere within the Audicle window, you can change your view of the spheres. When shreds are active in the virtual machine, their corresponding spheres move in the *plane of shredular existence*. Once a shred finishes (or is removed from the VM), it leaves and floats away into the distance (Figure 5.8).

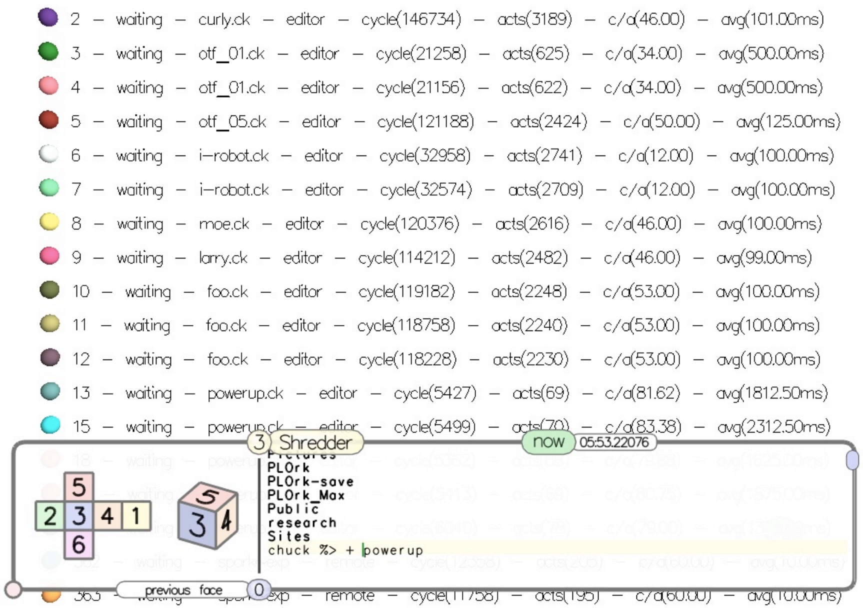
By clicking on the green sphere in the bottom right of the Audicle window, the programmer can view a textual list of shreds (both active and finished), as well as real-time statistics about each shred (Figures 5.9 and 5.10). They include the following information.

- the identification number assigned to the shred by the virtual machine.
- whether a shred is currently waiting or running. (Since the model in ChucK is to compute between advancing time, it can be extremely rare to "catch" a shred in a *running* state – unless it's failing to advance time. This can be useful for identifying hanging shreds)
- where the code came from (e.g., from buffer, file, or network).
- number of VM bytecode instructions executed.
- number of times the shred has advanced time, also called *activations*.
- a ratio of executed VM instructions to activations.
- dynamically computed average control rates.



```
Pictures
PLOrk
PLOrk-save
PLOrk_Max
Public
research
Sites
chuck %> + powerup
```

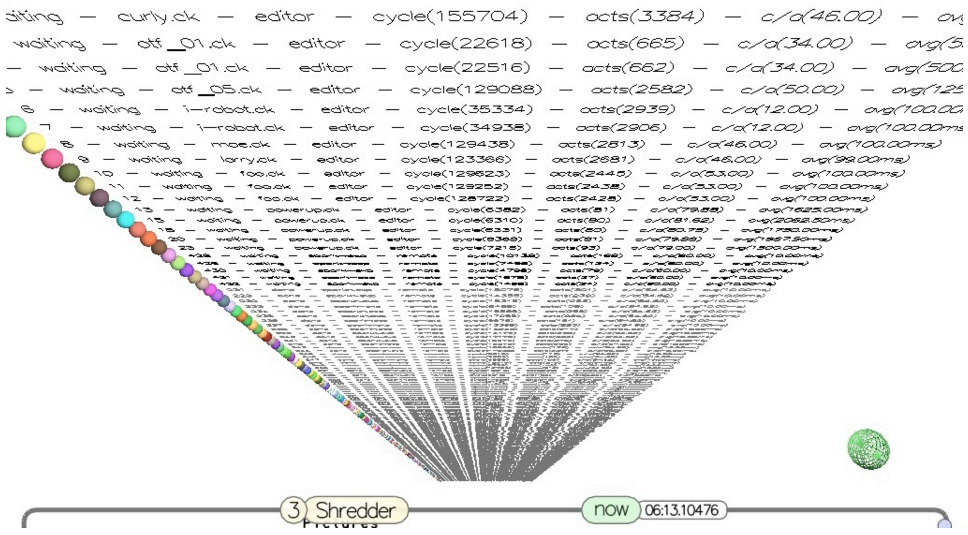
Figure 5.8: The Shredder: visualizing active and deactivated shreds (the latter ascending towards viewer.



```
● 2 – waiting – curly.ck – editor – cycle(146734) – acts(3189) – c/a(46.00) – avg(101.00ms)
● 3 – waiting – otf_01.ck – editor – cycle(21258) – acts(625) – c/a(34.00) – avg(500.00ms)
● 4 – waiting – otf_01.ck – editor – cycle(21156) – acts(622) – c/a(34.00) – avg(500.00ms)
● 5 – waiting – otf_05.ck – editor – cycle(121188) – acts(2424) – c/a(50.00) – avg(125.00ms)
● 6 – waiting – i–robot.ck – editor – cycle(32958) – acts(2741) – c/a(12.00) – avg(100.00ms)
● 7 – waiting – i–robot.ck – editor – cycle(32574) – acts(2709) – c/a(12.00) – avg(100.00ms)
● 8 – waiting – moe.ck – editor – cycle(120376) – acts(2616) – c/a(46.00) – avg(100.00ms)
● 9 – waiting – larry.ck – editor – cycle(114212) – acts(2482) – c/a(46.00) – avg(99.00ms)
● 10 – waiting – foo.ck – editor – cycle(119182) – acts(2248) – c/a(53.00) – avg(100.00ms)
● 11 – waiting – foo.ck – editor – cycle(118758) – acts(2240) – c/a(53.00) – avg(100.00ms)
● 12 – waiting – foo.ck – editor – cycle(118228) – acts(2230) – c/a(53.00) – avg(100.00ms)
● 13 – waiting – powerup.ck – editor – cycle(5427) – acts(69) – c/a(81.62) – avg(1812.50ms)
● 15 – waiting – powerup.ck – editor – cycle(5499) – acts(70) – c/a(83.38) – avg(2312.50ms)
● 363 – waiting – spork–exp – remote – cycle(11758) – acts(195) – c/a(60.00) – avg(10.00ms)
```

```
Pictures
PLOrk
PLOrk-save
PLOrk_Max
Public
research
Sites
chuck %> + powerup
```

Figure 5.9: The Shredder: an Audicle face to visualize and monitor shreds.



```
aiting — curly.ck — editor — cycle(155704) — acts(3384) — c/a(46.00) — av
waiting — otf_01.ck — editor — cycle(22618) — acts(665) — c/a(34.00) — avg(5
— waiting — otf_01.ck — editor — cycle(22516) — acts(662) — c/a(34.00) — avg(500
— waiting — otf_05.ck — editor — cycle(129088) — acts(2582) — c/a(50.00) — avg(125
6 — waiting — i-robot.ck — editor — cycle(35334) — acts(2939) — c/a(12.00) — avg(100.00
7 — waiting — i-robot.ck — editor — cycle(34938) — acts(2906) — c/a(12.00) — avg(100.00ms
8 — waiting — moe.ck — editor — cycle(129438) — acts(2813) — c/a(46.00) — avg(100.00ms)
9 — waiting — larry.ck — editor — cycle(123366) — acts(2681) — c/a(46.00) — avg(99.00ms)
10 — waiting — foo.ck — editor — cycle(129623) — acts(2445) — c/a(53.00) — avg(100.00ms)
11 — waiting — foo.ck — editor — cycle(129252) — acts(2438) — c/a(53.00) — avg(100.00ms)
12 — waiting — foo.ck — editor — cycle(128722) — acts(2428) — c/a(53.00) — avg(100.00ms)
13 — waiting — powerup.ck — editor — cycle(6382) — acts(81) — c/a(79.88) — avg(1625.00ms)
15 — waiting — powerup.ck — editor — cycle(6310) — acts(80) — c/a(81.62) — avg(2062.50ms)
```

Figure 5.10: The Shredder: a top-down view.

5.3.4 Time and Timing

The Time 'n' Timing (or TnT) face of the Audicle visualizes the real-time temporal interactions of active shreds (Figure 5.11). Each row corresponds to a shred, and each vertical spike represents when a shred "wakes up", or is activated, to compute. Using this face, it's possible to gain an understanding of the relative timing of shreds, without necessarily having to look at the code.

5.3.5 Tabula Rasa

The original design of the Audicle included the Tabula Rasa face. Conceptually, this face was intended to be a "blank slate" for rendering custom real-time graphics, as specified in ChucK code. As of this writing, this face is currently unimplemented.

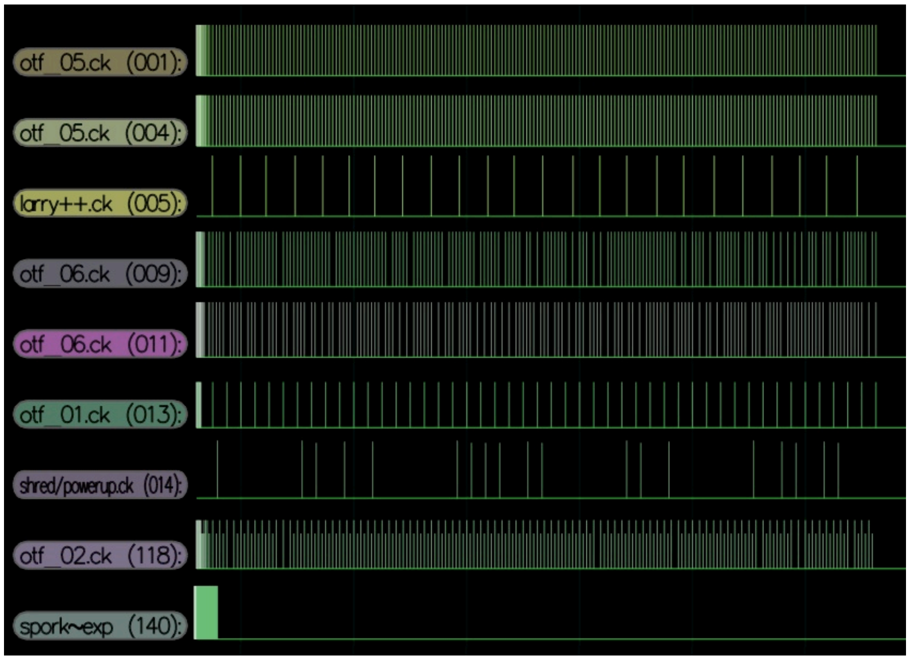


Figure 5.11: Time 'n' Timing (TNT): Audicle face to visualize relative timing between shreds.

5.4 Audicle Implementation

The Audicle's implementation consists of a graphical rendering engine, an I/O and networking system, a minimal windowing system, and internal logic with interface into the ChucK virtual machine. The implementation is in C/C++, with some high-level components written in ChucK. All components run in the same address space.

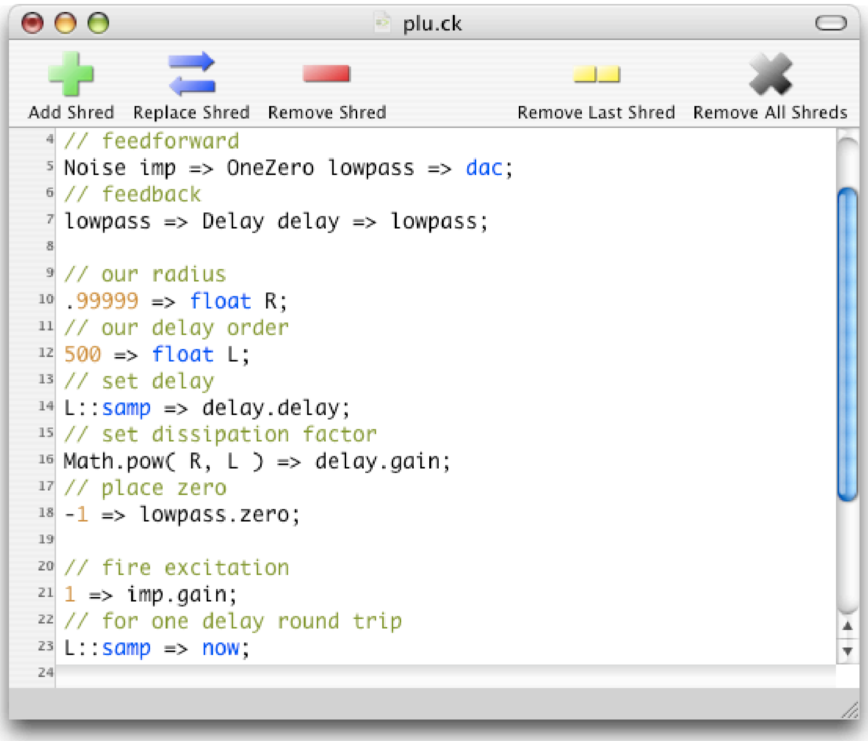
The graphics-rendering engine of the Audicle (implemented in the OpenGL API) runs on Mac OS X, Linux, and Windows. Using 3-D graphics with real-time audio synthesis is feasible and can be highly beneficial. With even modest graphics hardware support, the vast majority of the rendering can take place on the GPU (graphics processing unit), leaving the vast majority of CPU cycles for audio-related tasks. Using custom-built, minimal user interface elements, we can handle user-interface events more efficiently than the windowing sub-system, and with potentially better responsiveness. Because the rendering-rate stays relatively constant (at 30+ frame/second), the CPU usage stays constant and is less subject to large bursts due to user interface processing. Also, 3D graphics is flexible. It can emulate 2D when needed, and also provides significant viewing freedom.

5.5 miniAudicle

While the Audicle presents several useful visualizations and interactions, it can be somewhat difficult to use for writing code in longer sessions. This motivated the *miniAudicle* [74], an GUI-oriented, integrated environment for developing programs using ChucK (Figure 5.12). miniAudicle features a text editor, an embedded virtual machine, a virtual machine monitor, a stdin/stderr monitor for displaying log and error messages from the virtual machine, a ChucK shell, and support for on-the-fly programming commands like add, remove and replace. miniAudicle also supports creation and usage of typical graphical user interface widgets directly from ChucK code, for modifying program behavior and parameters at runtime. Currently, the miniAudicle uses the Cocoa API in the Mac OS X operating system, and WxWidgets under Windows and Linux, to render its graphical user interface.

5.6 Discussion

The Audicle is intended to be at once a development environment, a runtime environment, a visualizer, and a *out-of-the-box* on-the-fly programming performance platform. (Andrew Appel once noted that it is "program monitoring as performance art".)



```
// feedforward
Noise imp => OneZero lowpass => dac;
// feedback
lowpass => Delay delay => lowpass;

// our radius
.99999 => float R;
// our delay order
500 => float L;
// set delay
L::samp => delay.delay;
// set dissipation factor
Math.pow( R, L ) => delay.gain;
// place zero
-1 => lowpass.zero;

// fire excitation
1 => imp.gain;
// for one delay round trip
L::samp => now;
```

Figure 5.12: miniAudicle: a lightweight integrated development environment for ChucK and on-the-fly programming.

On-the-fly programming opens the potential for interesting interactions and visualizations in the audio programming process. Through the different faces in the Audicube, the programmer, composer, and performer can develop code in a version-tracking editor, and simultaneously visualize its behavior in terms of concurrency, timing, and runtime interactions with the rest of the system.

The integrated, on-the-fly environment of the Audicle helps to complete the development-runtime loop. The expressive power of coding is made available for runtime manipulation. In turn, on-the-fly information from runtime aids the development process, refining the powers of both. We gain the advantages of immediate feedback in an ever-modifiable continuum.

Additionally, the Audicle encourages the rapid prototyping mentality where continuous exploration and experimentation are valued. The Audicle further motivates the notion of runtime programmability as a form of performance, where code is used to expressively control audio synthesis/analysis and the process is conveyed to the audience. It also provides a platform where a degree of virtuosity can evolve. Due to its visual nature and immediate feedback, the Audicle can also be a useful compositional environment, where the composer can incrementally work on concurrent parts of a program piece. Similarly, it could function as an educational tool for teaching synthesis, audio programming, and multimedia. Potentially, the Audicle is the beginning of a new class of environments for developing programs on-the-fly, as well as for visualizing the audio programming process.

We look forward to experimenting with new interfaces for on-the-fly editing and code control, and new types of visualizations. Also, future work might investigate the technical and aesthetic aspects of collaborations between remotely connected *Audiclae* (plural form of Audicle), as well as new on-the-fly programming systems and environments (see Future Works in the Conclusion chapter).

Chapter 6

Applications and Evaluations

The ChucK programming language has found a variety of applications in composition, performance, sound design, research, and pedagogy – and continues to explore new areas of use. This chapter describes a number of endeavors to which ChucK has been applied, and evaluates the impact and effectiveness in those cases.

To provide a context for some of these applications, and for those that are curious, we first trace the evolution of the ChucK language and its various milestones in development and use. Section 2 discusses using ChucK as a teaching tool, particularly in the Princeton Laptop Orchestra, Stanford University, and other institutions. Section 3 examines ChucK as a software tool for composition and real-time performance, focusing on works in on-the-fly programming, sound design, building and testing interactive systems, and again in the context of the laptop orchestra. Some pieces created using ChucK are described. Feedback from the community and from students are presented. Finally, Section 4 discusses additional applications.

6.1 Evolution of ChucK

The ChucK programming language has evolved rapidly since its inception in 2002 (Figure 6.1). It was first presented at the 2003 International Computer Music Conference in Singapore [95]; the first ChucK live coding performance occurred shortly thereafter. ChucK source code (version 1.1, codename *Frankenstein*) was released under the General Public License (GPL) in Summer of 2004. In the same year, on-the-fly programming using ChucK and the Audicle were also introduced [97, 98].

2005 (Figure 6.2) saw a redesign of the language which supported arrays, object-oriented programming, events, as well as support for Open Sound Control [104], controller-mapping [102], and human-interface devices (HID). ChucK/Audicle were being presented in various venues (Vancouver, Barcelona, Rome, Beijing) and the development team had grown to more than a dozen programmers, testers, and documenters. In the fall of 2005, Princeton Laptop Orchestra (PLOrk) kicked off, founded by Dan Trueman and Perry Cook, and developed and instructed by Trueman, Cook, Scott Smallwood, and the author. This provided a intense platform for ChucK pedagogy, development, and deployment in compositions and performances.

The years spanning 2006 and 2008 (Figure 6.3 and 6.4) witnessed an intensified usage of ChucK both at Princeton University, Stanford University (where the author started on the faculty of the Center for Computer Research for Music and Acoustics (CCRMA) in 2007, while continuing to write this document), and in the community at-large.

New features continued to make their way into the language, and new experiments were carried out, including building graphical interfaces in the Audicle coupled to ChucK code for dealing with topics such as sound synthesis, audio DSP

[19, 80, 65], networking [102], and the integration of ChucK into audio analysis/synthesis frameworks, notably TAPESTREA [62]. Live performances took place at different scales and venues, ranging from solo and duo performances to orchestras consisting of 15 to 20 laptops.

6.2 Teaching ChucK

Many opportunities have been taken to make use of and assess ChucK as a pedagogical tool for sound synthesis/analysis, physical modeling, programming, computer-mediated instrument design, and live performance. As the timelines (Figures 6.1, 6.2, 6.3, 6.4) document, ChucK has been presented to different types of audiences – in conference presentations, workshops, demonstrative performances, and courses. By far, however, our most intense and first-hand user study and experience with teaching ChucK first came with the Princeton Laptop Orchestra (PLOrk) [103], and subsequently at CCRMA and in the Stanford Laptop Orchestra [94]. The results suggest that ChucK is highly effective in teaching both audio/music and programming to experienced and novice programmers alike.

This section documents the experiences of teaching ChucK at Princeton and other institutions, and evaluates the results. We begin with ChucK in the Princeton Laptop Orchestra and then move to other settings. We examine and evaluate the laptop orchestra classroom and its approaches for teaching, especially in the context of ChucK. In doing so, we describe an integrated, naturally interdisciplinary teaching/learning environment for computer science, music, and performance. In such an environment, the learning and internalization of technical knowledge happens symbiotically with the acquisition of aesthetic and artistic awareness; there

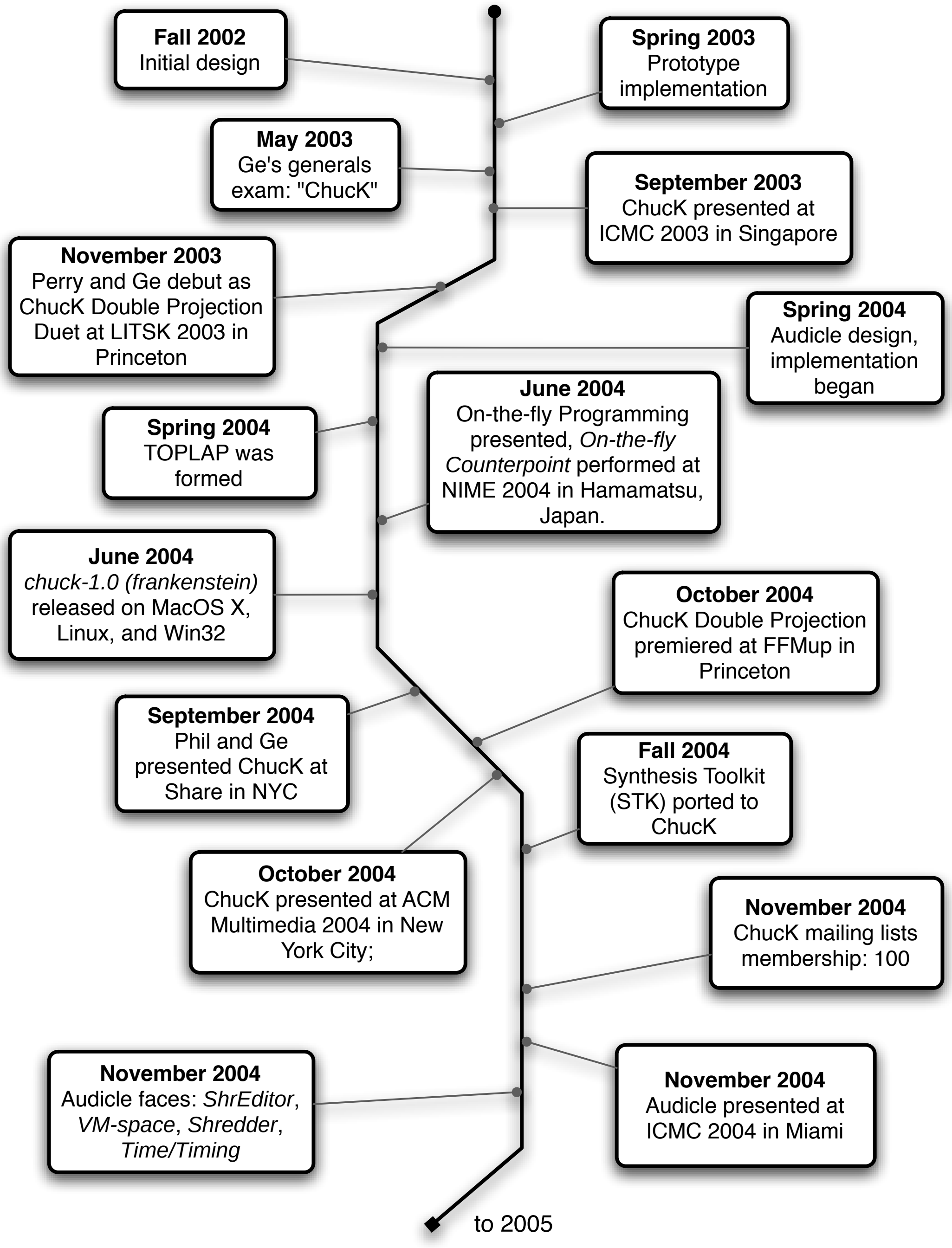

Fall 2002
Initial design
Spring 2003
Prototype implementation
May 2003
Ge's generals exam: "ChucK"
September 2003
ChucK presented at ICMC 2003 in Singapore
November 2003
Perry and Ge debut as ChucK Double Projection Duet at LITSK 2003 in Princeton
Spring 2004
Audicle design, implementation began
Spring 2004
TOPLAP was formed
June 2004
On-the-fly Programming presented, *On-the-fly Counterpoint* performed at NIME 2004 in Hamamatsu, Japan.
June 2004
chuck-1.0 (frankenstein) released on MacOS X, Linux, and Win32
October 2004
ChucK Double Projection premiered at FFMup in Princeton
September 2004
Phil and Ge presented ChucK at Share in NYC
Fall 2004
Synthesis Toolkit (STK) ported to ChucK
October 2004
ChucK presented at ACM Multimedia 2004 in New York City;
November 2004
ChucK mailing lists membership: 100
November 2004
Audicle faces: *ShrEditor*, *VM-space*, *Shredder*, *Time/Timing*
November 2004
Audicle presented at ICMC 2004 in Miami
to 2005

Figure 6.1: Timeline: evolution of ChucK, 2002-2004.

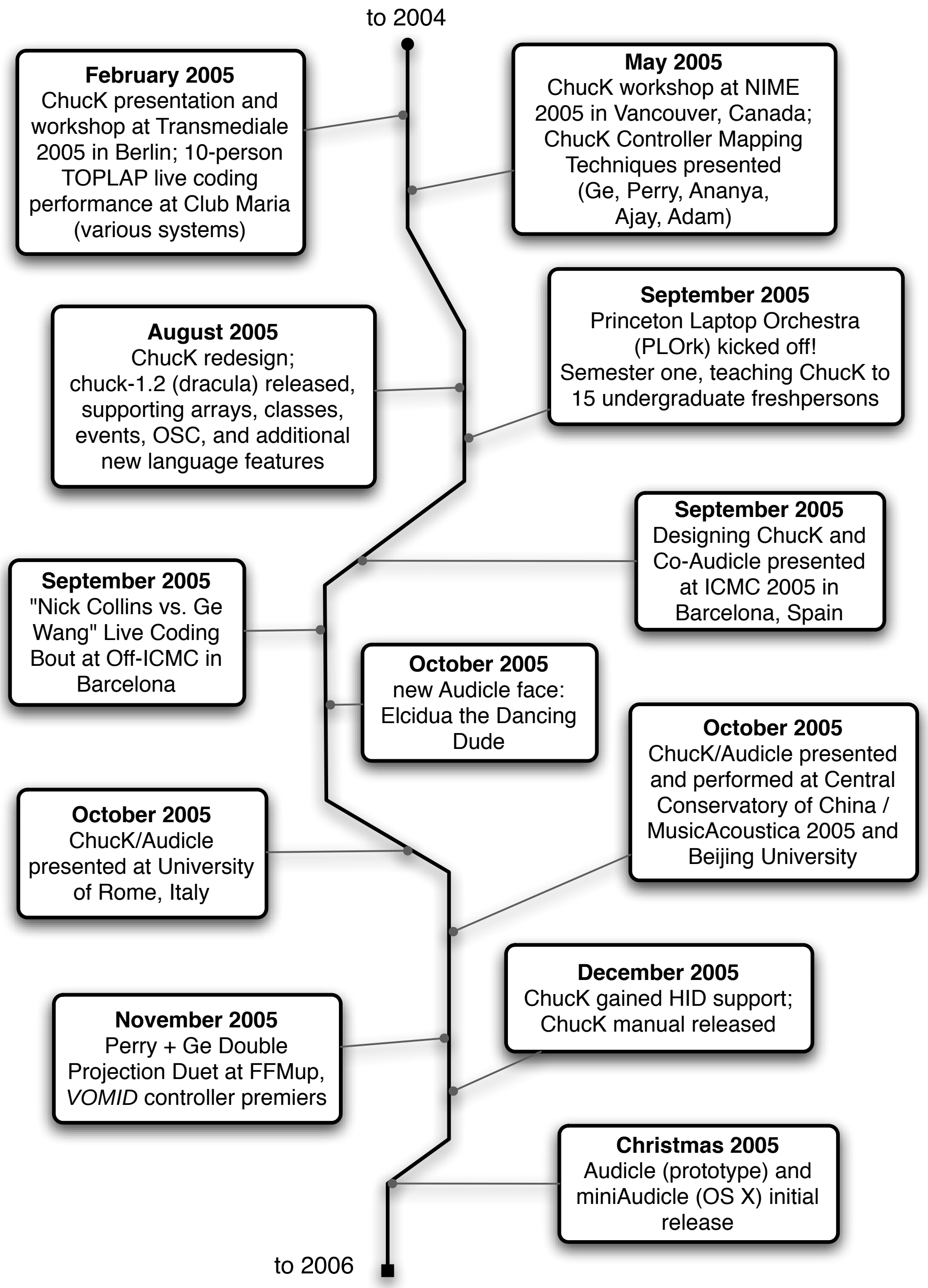

to 2004
February 2005
ChucK presentation and workshop at Transmediale 2005 in Berlin; 10-person TOPLAP live coding performance at Club Maria (various systems)
May 2005
ChucK workshop at NIME 2005 in Vancouver, Canada; ChucK Controller Mapping Techniques presented (Ge, Perry, Ananya, Ajay, Adam)
August 2005
ChucK redesign; chuck-1.2 (dracula) released, supporting arrays, classes, events, OSC, and additional new language features
September 2005
Princeton Laptop Orchestra (PLOrk) kicked off! Semester one, teaching ChucK to 15 undergraduate freshpersons
September 2005
Designing ChucK and Co-Audicle presented at ICMC 2005 in Barcelona, Spain
September 2005
"Nick Collins vs. Ge Wang" Live Coding Bout at Off-ICMC in Barcelona
October 2005
new Audicle face: Elcidua the Dancing Dude
October 2005
ChucK/Audicle presented and performed at Central Conservatory of China / MusicAcoustica 2005 and Beijing University
October 2005
ChucK/Audicle presented at University of Rome, Italy
December 2005
ChucK gained HID support; ChucK manual released
November 2005
Perry + Ge Double Projection Duet at FFMup, *VOMID* controller premiers
Christmas 2005
Audicle (prototype) and miniAudicle (OS X) initial release
to 2006

Figure 6.2: Timeline: evolution of ChucK, 2005.

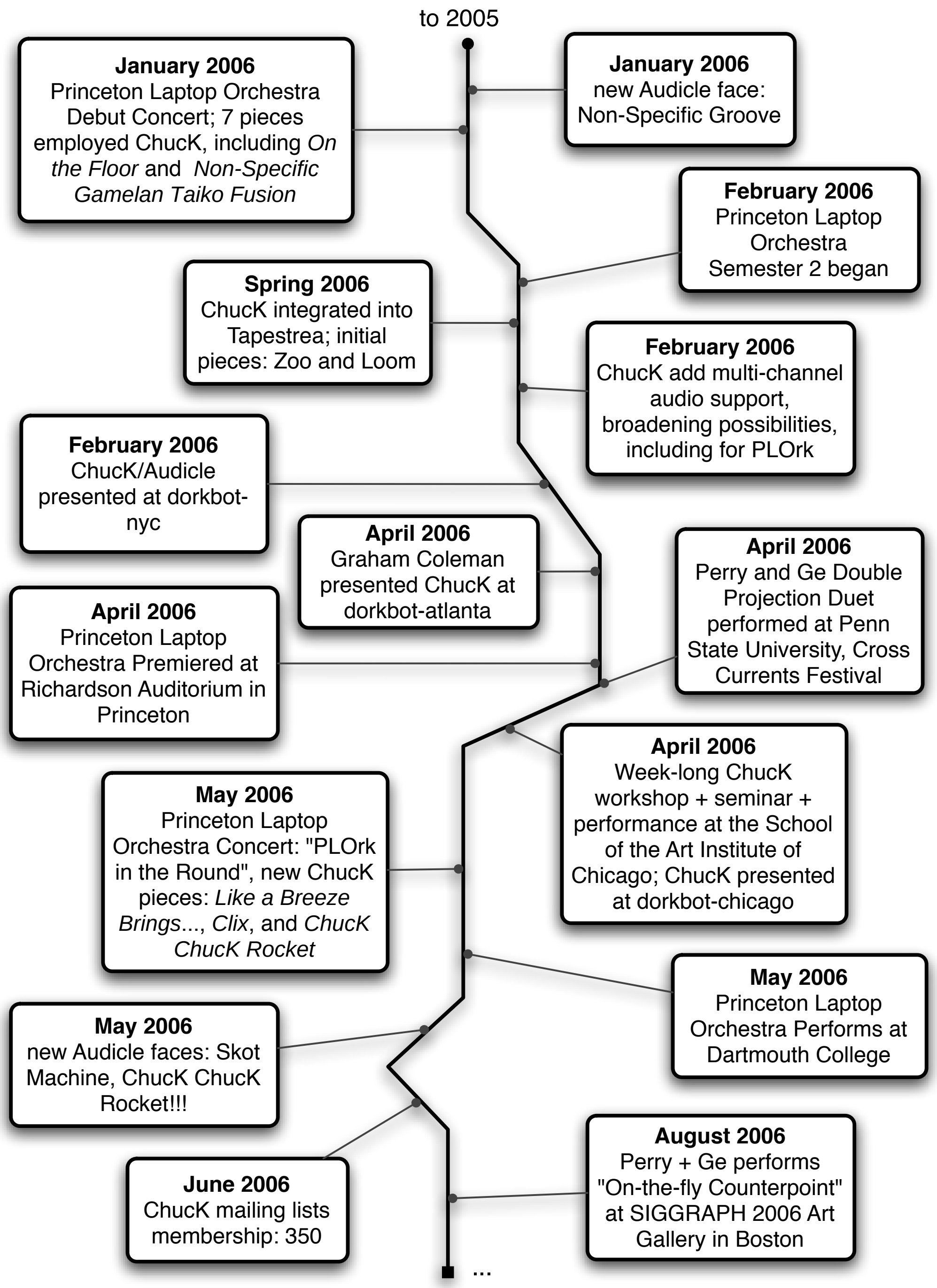

to 2005
January 2006 Princeton Laptop Orchestra Debut Concert; 7 pieces employed ChucK, including *On the Floor* and *Non-Specific Gamelan Taiko Fusion*
January 2006 new Audicle face: Non-Specific Groove
February 2006 Princeton Laptop Orchestra Semester 2 began
Spring 2006 ChucK integrated into Tapestrea; initial pieces: Zoo and Loom
February 2006 ChucK add multi-channel audio support, broadening possibilities, including for PLOrk
February 2006 ChucK/Audicle presented at dorkbot-nyc
April 2006 Graham Coleman presented ChucK at dorkbot-atlanta
April 2006 Perry and Ge Double Projection Duet performed at Penn State University, Cross Currents Festival
April 2006 Princeton Laptop Orchestra Premiered at Richardson Auditorium in Princeton
April 2006 Week-long ChucK workshop + seminar + performance at the School of the Art Institute of Chicago; ChucK presented at dorkbot-chicago
May 2006 Princeton Laptop Orchestra Concert: "PLOrk in the Round", new ChucK pieces: *Like a Breeze Brings...*, *Clix*, and *ChucK ChucK Rocket*
May 2006 Princeton Laptop Orchestra Performs at Dartmouth College
May 2006 new Audicle faces: Skot Machine, ChucK ChucK Rocket!!!
August 2006 Perry + Ge performs "On-the-fly Counterpoint" at SIGGRAPH 2006 Art Gallery in Boston
June 2006 ChucK mailing lists membership: 350
...

Figure 6.3: Timeline: evolution of ChucK, 2006.

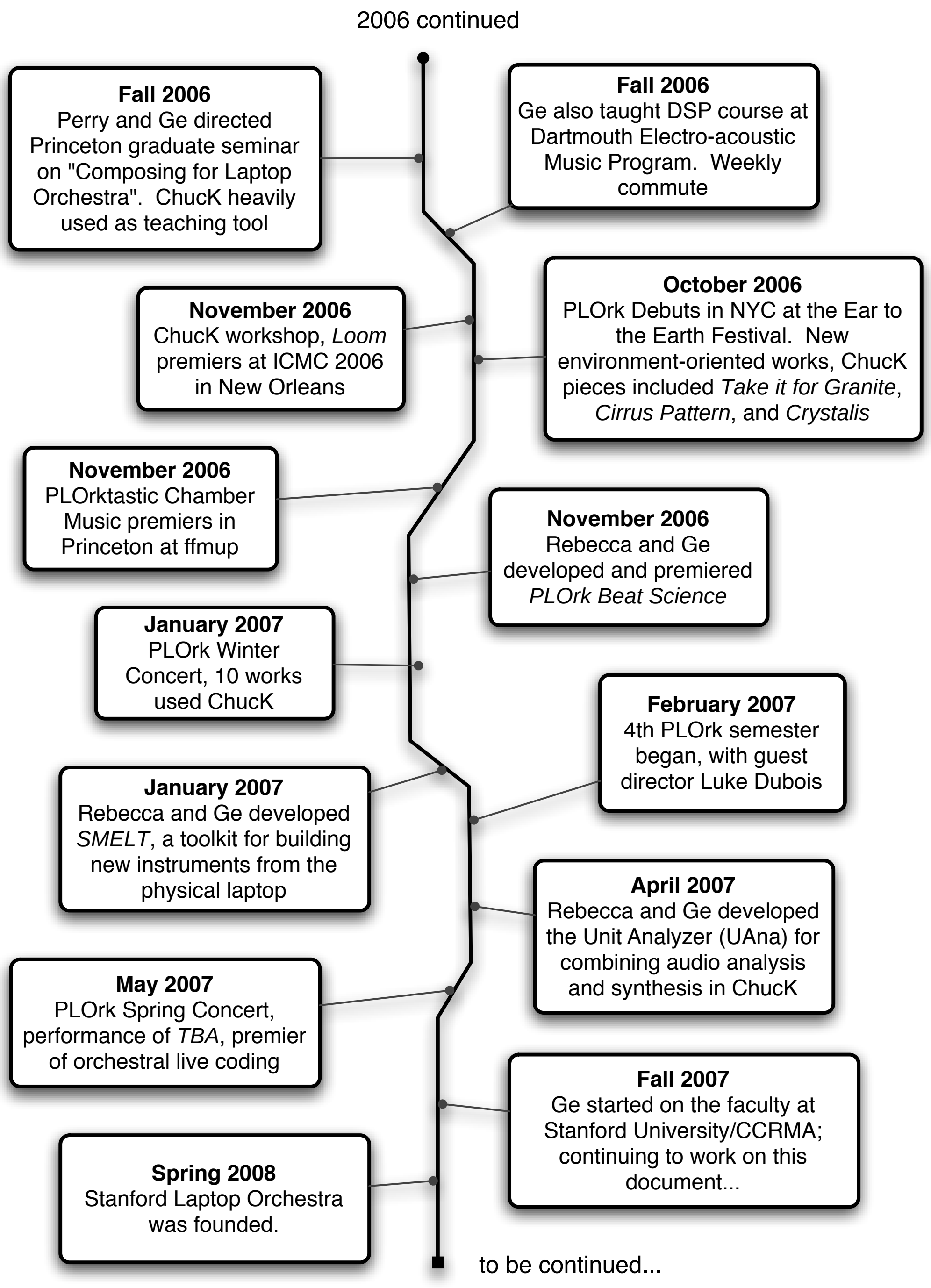

2006 continued
Fall 2006
Perry and Ge directed Princeton graduate seminar on "Composing for Laptop Orchestra". ChucK heavily used as teaching tool
Fall 2006
Ge also taught DSP course at Dartmouth Electro-acoustic Music Program. Weekly commute
November 2006
ChucK workshop, *Loom* premiers at ICMC 2006 in New Orleans
October 2006
PLOrk Debuts in NYC at the Ear to the Earth Festival. New environment-oriented works, ChucK pieces included *Take it for Granite*, *Cirrus Pattern*, and *Crystalis*
November 2006
PLOrktastic Chamber Music premiers in Princeton at ffmup
November 2006
Rebecca and Ge developed and premiered *PLOrk Beat Science*
January 2007
PLOrk Winter Concert, 10 works used ChucK
February 2007
4th PLOrk semester began, with guest director Luke Dubois
January 2007
Rebecca and Ge developed *SMELT*, a toolkit for building new instruments from the physical laptop
April 2007
Rebecca and Ge developed the Unit Analyzer (UAna) for combining audio analysis and synthesis in ChucK
May 2007
PLOrk Spring Concert, performance of *TBA*, premier of orchestral live coding
Fall 2007
Ge started on the faculty at Stanford University/CCRMA; continuing to work on this document...
Spring 2008
Stanford Laptop Orchestra was founded.
to be continued...

Figure 6.4: Timeline: evolution of ChucK, 2006-2008.

is only one explicit goal: learn to make compelling computer-mediated music together in an academic setting; all other learning happens "along the way" or "by accident". We believe this is an exciting new environment where the learning of interdisciplinary knowledge is not only natural, but also inevitable (and fun).

6.2.1 Princeton Laptop Orchestra

In fall of 2005, PLOrk commenced in its inaugural semester, instructed by Dan Trueman, Perry Cook, Scott Smallwood, and the author [86, 85, 79, 103]. This first-of-its-kind ensemble and course consisted of 15 independent laptop/six-channel speaker array stations programmed and operated by 15 undergraduates freshmen, to whom we taught Max/MSP and ChucK. The students entered the class with no prior programming experience. However, over the following 4 months, we covered topics ranging from sound synthesis/design to programming to controller mapping and live computer-mediated performances. The students took very well to ChucK from the beginning, and while we did not formally cover advanced topics (such as object-oriented programming), all the students internalized the language to the point they can comfortably focus on creating compositions and performances.

For example, an early assignment (3rd week) asked the students to build a generative drum machine using ChucK, employing the time control and concurrency mechanisms in the language, and perform it using on-the-fly programming. We were extremely pleased to discover the students delivered quality works that demonstrated both technical comprehension and creative zeal. Subsequent assignments, including trio/duo performances, creating a soundscape, met with similar enthusiasm and success. Descriptions of some representative ChucK assignments are reproduced in the next section.

Figure 6.5: PLOrk class in session.

Much of the success was undoubtedly due to the sheer creative will and energy of the students (they all were fantastic), at the same time it also demonstrated that ChucK can be a viable teaching tool. Below is an encouraging quote (reproduced from the README to the drum machine assignment) from Anna, a student in the first PLOrk class (and a talented cellist):

> "... However, when everything worked the way it was supposed to, when my spontaneous arrangement of computer lingo transformed into a musical composition, it was a truly amazing experience. The ability to control duration and pitch with loops, integers, and frequency notation sent me on a serious power trip."

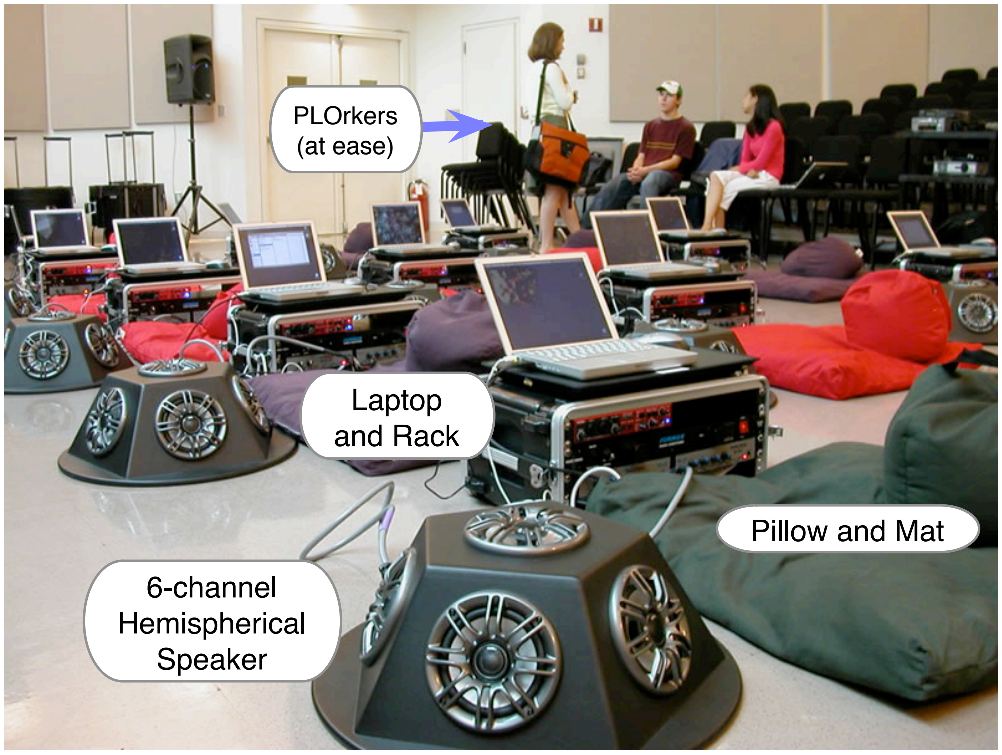


Figure 6.6: PLOrk setup (individual stations).

It wasn't all smooth sailing, of course, a multitude of bugs/features were discovered, and a lot of on-the-fly fixing took place. But no one got discouraged, and we introduced a lot of features and bug fixes (and new bugs) as a result. Also, we taught topics using Max/MSP at the same time, which turned out to be pedagogically fruitful, since it exposed two drastically different paradigms. This was also practically because Max and ChucK tended to crash in different places, and having several options for the task-at-hand is usually good.

6.2.2 Assignments

In this subsection, we document some representative project specifications based on ChucK, assigned to students in the laptop orchestra.

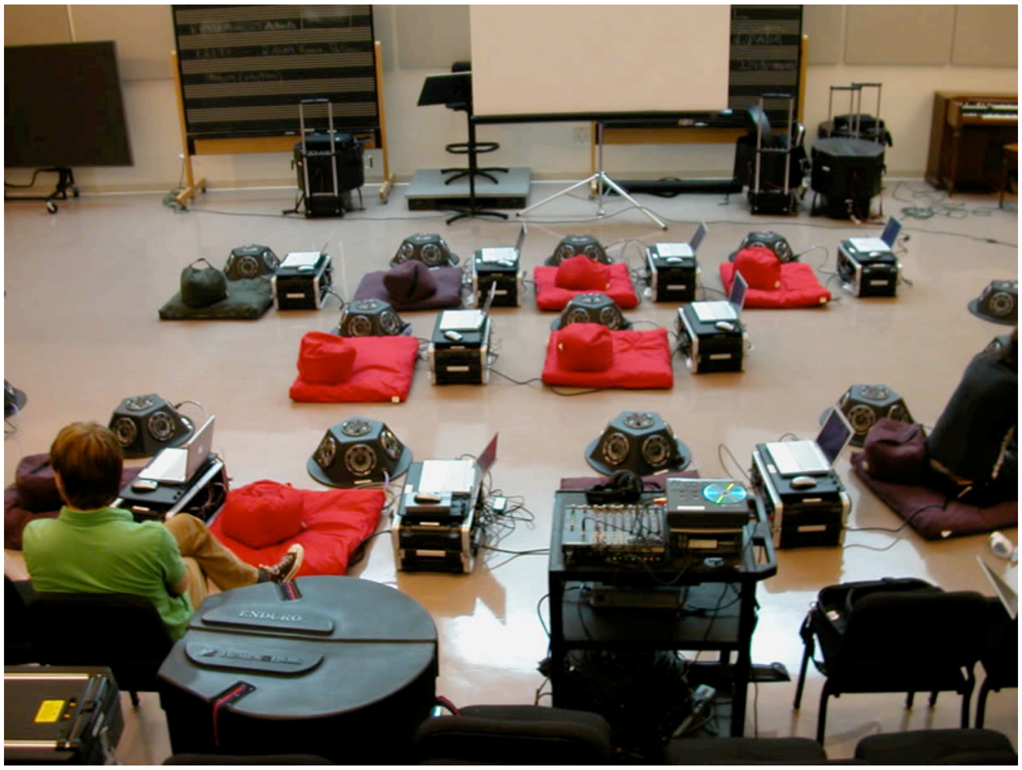


Figure 6.7: PLOrk setup (minus humans).

Play with ChucK!

In this first assignment, the goal was to ease everyone into the language and environment and to provide a creative space in which to experiment. The assignment read thus:

1. Run the examples given (don't hesitate to post/email questions).
2. Open up a few programs (try using TextEdit on OS X, or WordPad on Windows) and get a general idea of the code. Try modifying some of the parameters, save the file (perhaps under a different name), and run it with ChucK. Does the result sound simlilar to what you expected?

3. Create two ChucK programs (of arbitrary length), either using one of the examples as a model or starting from scratch; One program should generate sound(s) or music that is "rhythmic". the other program should generate sound(s) or music that has little or no "rhythm". you define what "rhythmic" means. We will play these programs together in class.

4. Turn in the programs with a short README file that explains the two programs you have created and any interesting experience or problems you encountered in their making.

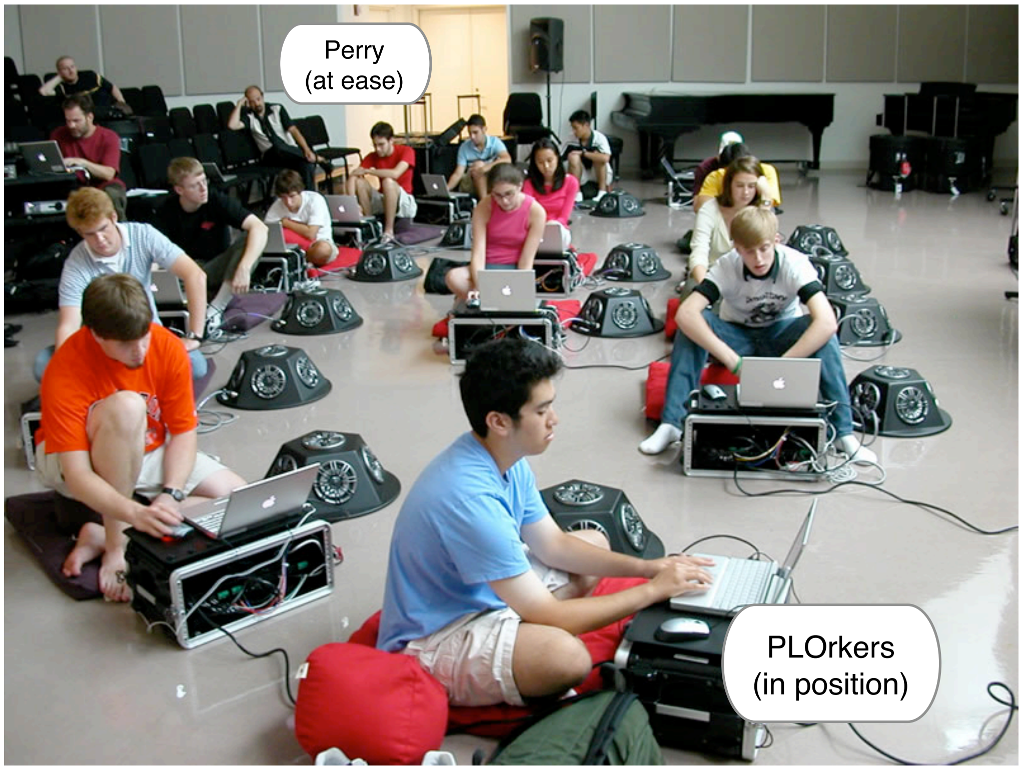


Figure 6.8: PLOrk in action.

Drum Machine

This second assignment turned out to be successful in allowing the participants to understand the main ideas of the language (time and concurrency) while operating within a familiar and creative framework.

1. Create a drum machine using multiple shreds - play them using on-the-fly commands

   (a) Experiment with playing the OTF examples using on-the-fly programming commands (+, -, =, --, etc).

   (b) Find drum samples (or samples of other percussive sounds); each sample should only be a single strike (and not a loop); you may need to edit

Figure 6.9: The Stanford Laptop Orchestra classroom in motion.

them (e.g., using Audacity or another sound editor); this need not take a long time, but do pay attention to individual and collective quality and feel of the sounds, as that will make a big difference in the final result.

(c) Put these files in a folder (your chuck programs will refer to these files).

(d) If possible, credit the source of the samples in your README.

(e) Make a shred for each drum sound (feel free to base them on the examples).

(f) Choose a tempo that all shreds should agree on (**.5::second => T;**).

(g) Synchronize to this period at top of each file (**T - (now % T) => now;**). In certain cases, it might make sense to control more than one drum sound in a shred (like if two sounds always go together). In general, however, you should split up the drum machine so that you can independently control each component.

(h) (optional) Add additional parts (drone, melody, bassline, etc) if you like, but that is not required - focus on getting the percussive parts done first.

(i) Practice playing the drum machine you created using on-the-fly programming commands (+, -, =, –, etc).

(j) Have fun!

2. Write a short README text file that describes what you did and any interesting problems or challenges you encountered.

Soundscape

- Create/compose an ambient soundscape using ChucK (this will further augment your arsenal of sounds for your final projects). The soundscape should be coherent and consistent within some environment/idea/framework of your choosing, for example:

- Cities of Earth 2049
- The Deep Ocean
- Violent industrialization
- Post-apocalypse landscape
- Charlie and The Chocolate/Hoagie Factory
- (invent your own)

- Guidelines:
  - incorporate at least 10 different components
    - can be mixture of synthesized or recorded sounds
    - can be long or short in duration (a mixture is often good)
    - soundfiles: use only short soundfiles (shorter than 5 seconds)
    - layer the soundscape in separate chuck programs
    - use OTF to control the content and the texture of the world you create
  - try to identify places in your programs where functions might be helpful (and use them)
  - score/script a 4-10 minute composition using these sounds (in text/graph/timeline or whatever else)
  - in your README, describe the unified environment/idea behind your soundscape
- Optional:

- interlocking rhythms
- harmonic and melodic ideas
- controllable via MIDI

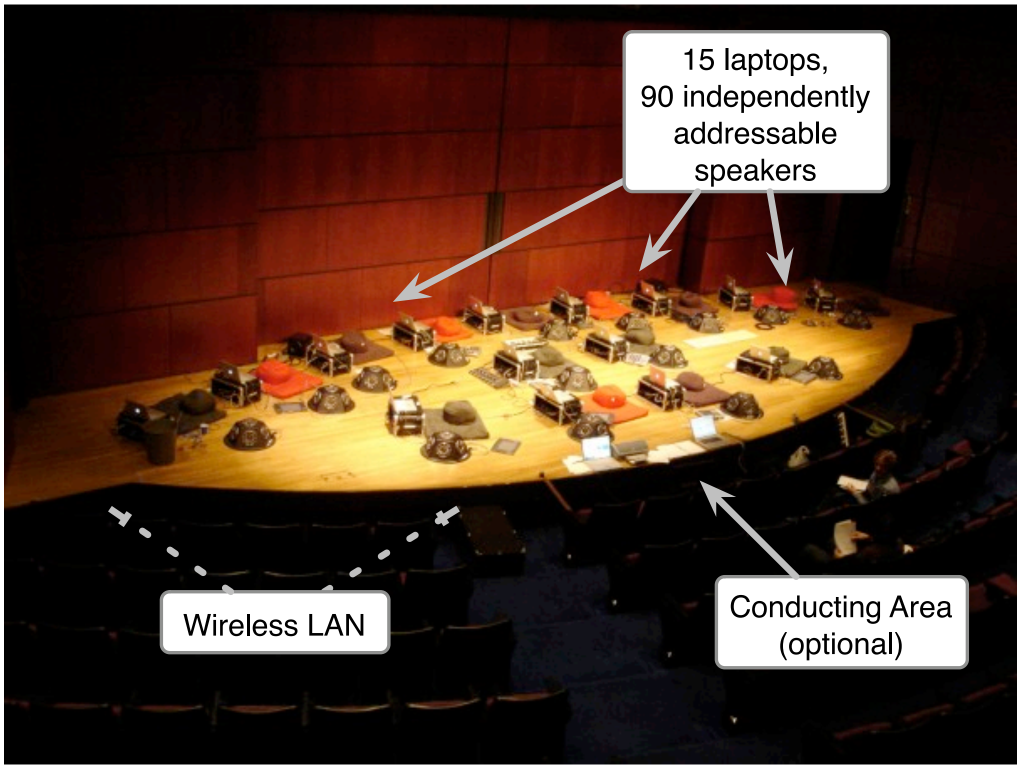

15 laptops,
90 independently addressable speakers
Wireless LAN
Conducting Area (optional)

Figure 6.10: PLOrk setup, onstage at Taplin Auditorium, Princeton.

Trio and Duo Performance

Twice in that first semester of the laptop orchestra, the students formed trio and duo ensembles to create and perform a computer-mediated work. The first occassion was in class, the second took place in the PLOrk Debut Concert in January 2006, in Taplin Auditorium in Princeton University. A great variety of software and hardware devices were employed, ranging from sensors (FSR's, floor tiles, accelerometers, light sensors), WACOM tablets [53], MIDI-based keyboards and drumpads, to software for sound synthesis, controller mapping, live audio processing of voice and acoustic instruments, on-the-fly programming, and networking via OSC. It was intense but immensely rewarding, as things tended to work to good effect.

6.2.3 Results and Evaluation

The following results and observations were concluded from teaching ChucK in PLOrk:

- All students became avid and skillful ChucK programmers. As mentioned before, advanced topics such as object-oriented programming and networking were not included in the early ChucK/PLOrk curriculum, though several students learned about them for various projects. Several students still actively use ChucK. At the time of this writing (2008), some of the initial students are now Juniors and graduating Seniors, still using ChucK. One John Fontein is completing his 6th semester in PLOrk, writing his senior thesis in Music using ChucK.

- The language turned out to be straightforward to learn, and can be highly effective in teaching programming concepts as well as sound synthesis. In fact, the two reinforced each other. Perhaps having a immediately perceivable audio/musical feedback while programming allowed the student to more easily focus the task as hand, instead of simply programming for learning programming's sake.

- No prior programming experience was required. Nearly all students came into the class without such experience and all become adept at programming.

- ChucK's time and concurrency model was natural to understand. Explicit manipulation of time in the language for generating sound proved to be a strong and useful mechanism that provided a crucial understanding of the relationship between time and sound in programming.

- The non-preemptive concurrent programming model was highly amenable to expressing parallelism without having to worry about traditional difficulties of preemptive concurrency – race conditions, deadlock, and nondeterminism. Coupled with the timing mechanism, it made the topic of concurrency manageable to teach and learn in an introductory course.

- Finally, we were able to teach the language without sacrificing flexibility or programming concepts. We covered the basics of procedural programming plus ChucK timing, concurrency, and sound synthesis.

Additionally, we found the following presentation order of topics to be reasonable and effective:

- types, variables, values
- operators
- control structures
- importance of time in audio programming
- functions
- concurrency
- event-driven programming
- (advanced topics) object-oriented programming
- (advanced topics) networking

Additional Feedback

Here are some additional quotes reproduced from README files turned in with assignments.

> "It was so exciting to figure out how to control the exact rhythm produced by the shred, and I started working out rhythmic patterns on scrap paper in the form of music notation and then transferring it mathematically to the shred composition itself.
>
> I really like the on-the-fly command system as well. It may have driven my roommate crazy, but I was definitely jamming the whole way through. The only real problem with this assignment was knowing when to stop and get on with the rest of my work. This is so much better than memorizing French verbs." — Anna

> "This project was extremely fun. I did not use delay statements; instead I used IF statements with each measure being 16 beats. This way I could choose which sounds to play at what times and it is much easier. I decided to use bongos, because they sound wicked. I put together some rhythms, played them together, found out that they were not synchronized, decided that it sound way better that way, and so on and so forth. There weren't any troubles at all creating these files. There are 11 in total." — Bryan

6.2.4 Additional Courses

During the time of this writing (2006-2008), several institutions have adopted ChucK into their teaching curriculum, including Princeton University (in Computer Science

Figure 6.11: Teaching in the Stanford Laptop Orchestra.

and Music), California Institute of the Arts, George Institute of Technology, McGill University, University of Victoria, UC Santa Barbara, and others. At Stanford University, several full courses have used ChucK as the primary software platform, including "Fundamentals of Computer-Generated Sound", "Compositional Algorithms", and "Composing, Coding, and Performance with Laptop Orchestra" (Figure 6.11).

6.3 ChucK in Performance and Research

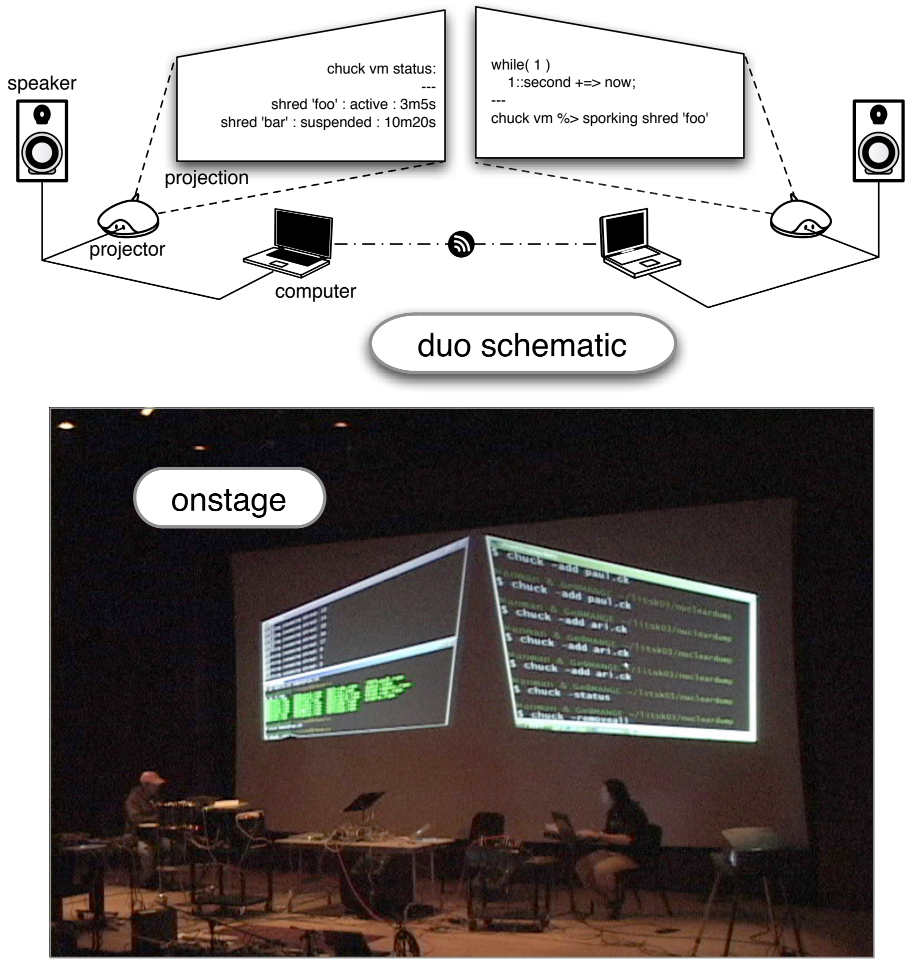
6.3.1 Performance in Laptop Orchestra

Below are some pieces composed, implemented, and performed in ChucK, including some interfaces built using the Audicle and ChucK.

On-the-fly Counterpoint "This piece (Figures 6.12, 6.13) is a study of the technical and aesthetic aspects of on-the-fly audio programming for live performance. The performers (Perry and Ge) use the ChucK language, which supports real-time, sample-synchronous, concurrent audio programming, and a highly on-the-fly style of programming, in which the composer / performer / programmer augments and modifies multiple programs while they are running, without stopping or restarting.

On-the-fly Counterpoint begins with a blank ChucK program. As part of the performance, we project the entire process on the screen for the audience to see and follow. We construct the counterpoint piece-by-piece in real-time, using the facets of concurrent audio programming and on-the-fly programming in ChucK. Contrapuntal simultaneities can be separated and compartmentalized into autonomous, concurrent entities. We can program and reason about each entity independently, as well as interact with other entities and with the program as a whole. This is part of our ongoing investigation into using code as an interactive and expressive musical instrument."

Non-Specific Gamelan Taiko Fusion Composer: Perry Cook and Ge Wang; conductor: Perry Cook. This piece is an experiment in human controlled, but machine synchronized (Figure 6.14) percussion ensemble performance. Various percussive sounds are temporally positioned by PLOrk members, and the piece gradually



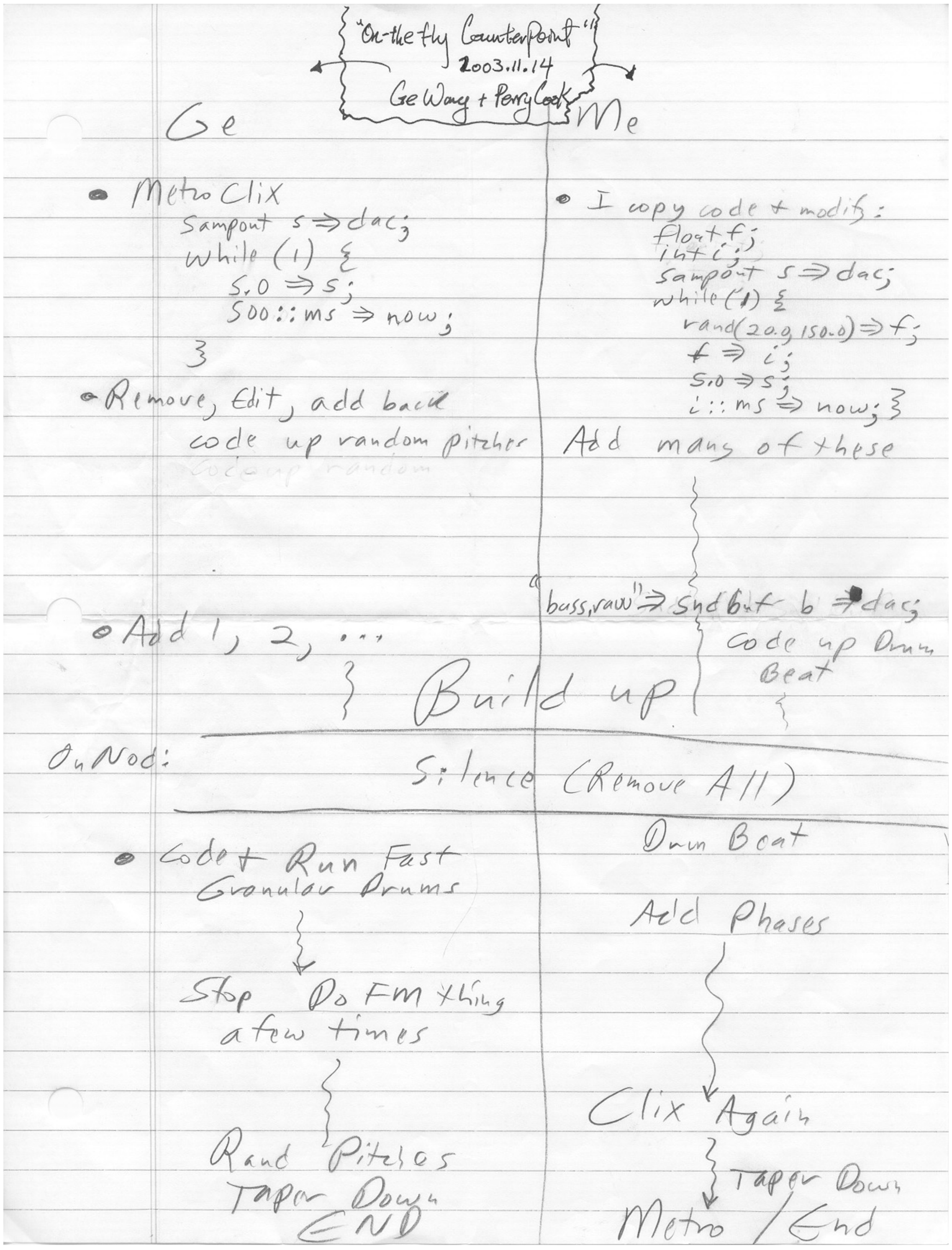
```
chuck vm status:
---
shred 'foo' : active : 3m5s
shred 'bar' : suspended : 10m20s
```

```
while( 1 )
    1::second +=> now;
---
chuck vm %> sporking shred 'foo'
```

Figure 6.12: A on-the-fly programming schematic.

transitions from tuned bell timbres to drums as the texture and density grows. The interface consists of a networked step sequencer implemented in the Audicle with ChucK as the audio engine (Figures 6.15, 6.16, 6.17).

CliX Composer and conductor: Ge Wang. In this piece, human operators type to make sounds, while their machines synthesize, synchronize, and spatialize the audio. Every key on the computer keyboard (upper/lower-case letters, numbers, symbols) is mapped to a distinct pitch (using the key's ASCII representation) and



"On-the-fly Counterpoint"
2003.11.14
Ge Wang + Perry Cook

Ge

• Metro Clix

```
sampout s => dac;
while (1) {
    5.0 => s;
    500::ms => now;
}
```

• Remove, Edit, add back
code up random pitches

• Add 1, 2, ...

} Build up

On Nod:

Silence (Remove All)

• Code + Run Fast
Granular Drums

Stop Do FM thing
a few times

Rand Pitches
Taper Down
END

Me

• I copy code + modify:

```
float f;
int i;
sampout s => dac;
while (1) {
    rand(20.0, 150.0) => f;
    f => i;
    5.0 => s;
    i::ms => now; }
```

Add many of these

"bass.raw" => sndbuf b => dac;
code up Drum Beat

Drum Beat

Add Phases

Clix Again

Taper Down

Metro / End

Figure 6.13: The score for *On-the-fly Counterpoint*.

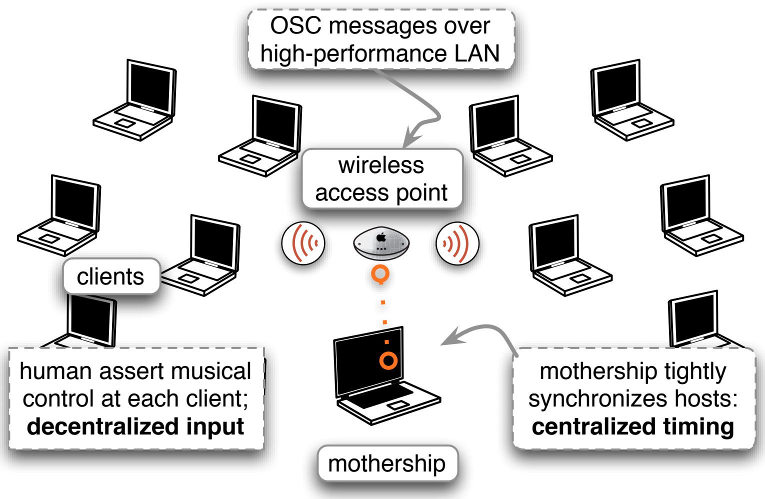

OSC messages over high-performance LAN
wireless access point
clients
human assert musical control at each client; **decentralized input**
mothership
mothership tightly synchronizes hosts: **centralized timing**

Figure 6.14: Network configuration (partial ensemble).

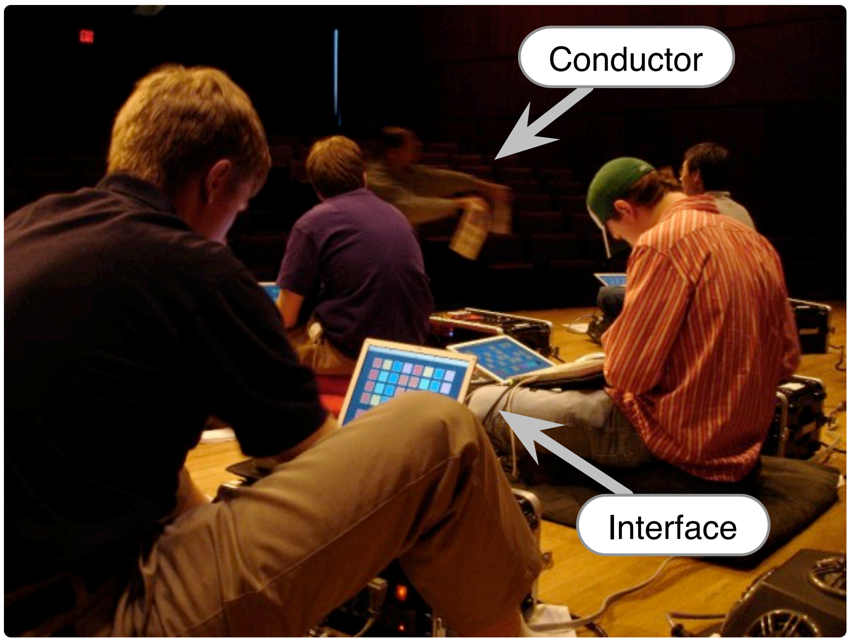


Figure 6.15: Non-Specific Gamelan Takio Fusion performed in PLOrk.

when pressed, emits a clicking sound that is synchronized in time to a common pulse. A (human) conductor coordinates frequency range, texture, movement, and timing (Figure 6.18).

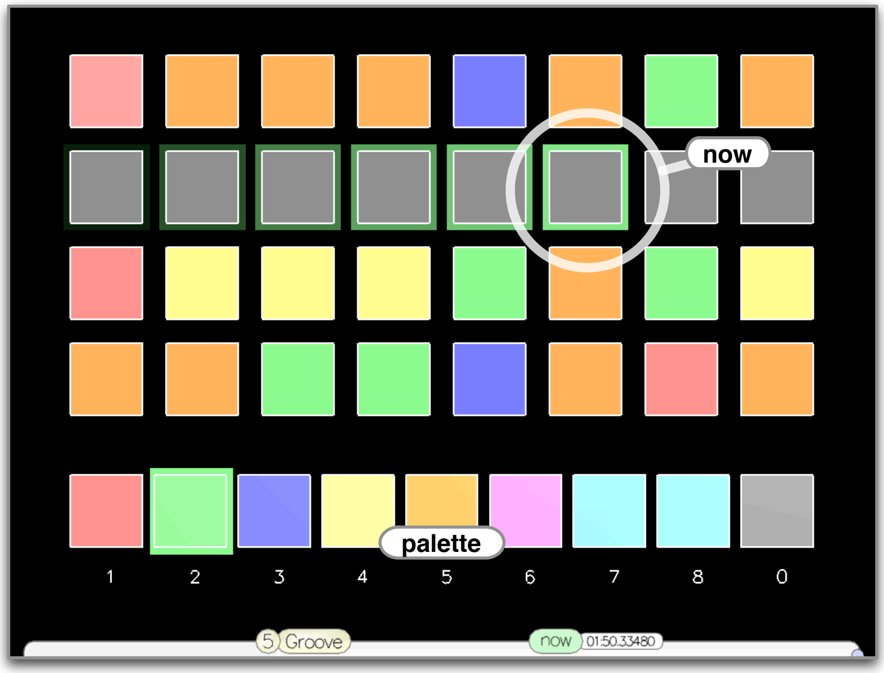


Figure 6.16: Non-Specific Groove: a network-synchronized colorful step sequencer implemented in the Audicle. The green highlight moves across the squares in real-time, as coordinated by the ensemble's master machine. Each color is associated with a different sound. All sound synthesis and networking are written in ChucK.

On the Floor Composer: Scott Smallwood. "You will notice when you walk into a casino that the machines are all tuned to the same key: a c-major chord. This chord floats around the space, in and out of every crevice, constantly arppeggiating, humming, droning, twittering echoing, sometimes incorporating snippets of melody. This happy drone soothes the nervous customers as they slowly drop their money into the machines. They create a sea of c-major, each and every one of them, pressing buttons on the machines, credit after credit, all day and all night." The virtual gambling interface was implemented in the Audicle (Figure 6.19).

a breeze brings... Composer: Scott Smallwood. "This *prelude* came about as a result of several mornings of hacking in ChucK. As I listened to the wind chimes

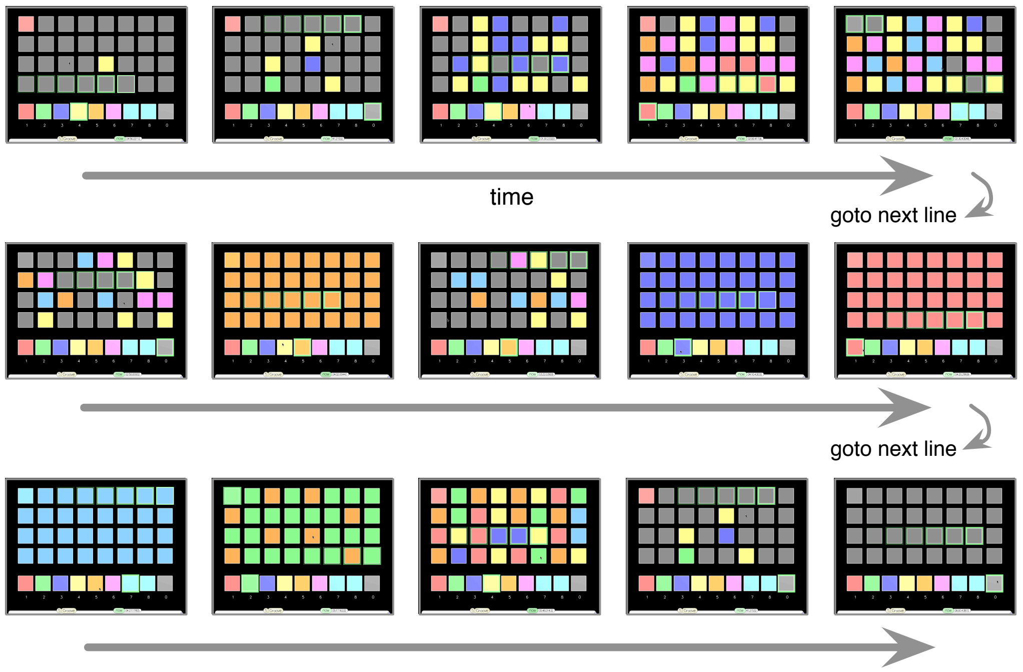


Figure 6.17: A possible sequence of suggested colors (texture) and density constitute the score, which the conductor visually conveys to the ensemble.

Figure 6.18: CliX in performance: the orchestra surrounds the audience (below) from around the balcony at Chancellor Green Library; conductor guides the direction of the performance.

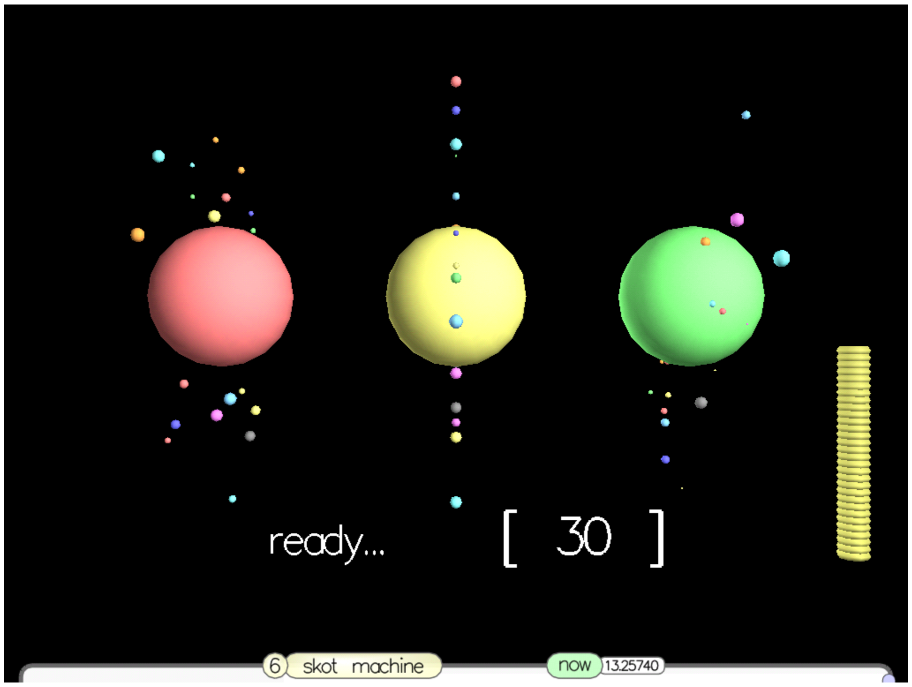


Figure 6.19: A interface for On The Floor, built in the Audicle, sound synthesis in ChucK.

outside my door, I began to realize that they were influencing the intuitive process of my experimentations. Before long I had created some algorithmic instruments that sounded rather nice together. This piece grows slowly out of the acoustic soundscape of the space, and then slowly subsides back into it, like a very slow breeze."

ChucK ChucK Rocket! Composers: Scott Smallwood and Ge Wang (additional graphics and animation programming by Ananya Misra). This game piece is a study that reflects our interest in creating games scenarios in which the sounds produced are part of an interactive sound composition. In this game, based on Chu Chu Rocket, mice are released onto a large grid. Each player has a piece of this grid, and is able to cause the running mice to change direction by placing arrows in their path, and they are also able to place objects in their path, which make sound when the mice run over them. Thus, a player can create a kind of instrument with their piece of the grid, trapping groups of mice into loops that contain sound objects of their choosing. They can also send mice to and receive mice from their neighbors through network portals, thus the mice are shared throughout the entire group. The interface was implemented in the Audicle, with ChucK as the audio engine (Figures 6.20 and 6.21).

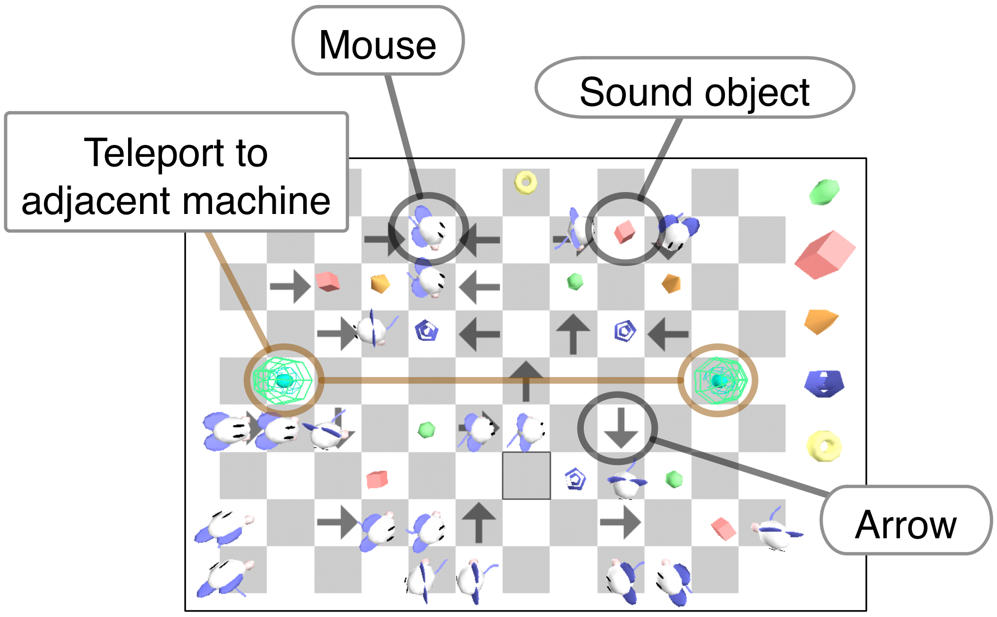


Figure 6.20: ChucK ChucK Rocket: game board as seen by one of the players.

Take it for Granite Composer: Perry Cook. "This sonic landscape was mined from recordings of stone sculptor Jonathan Shor's working of a large piece of granite. Perry recorded him drilling, placing shims, tapping the shims, and the wonderful sound of millions of years of energy being released as the stones split. The PLOrk players manipulate these sounds via a ChucK program that allows them to change

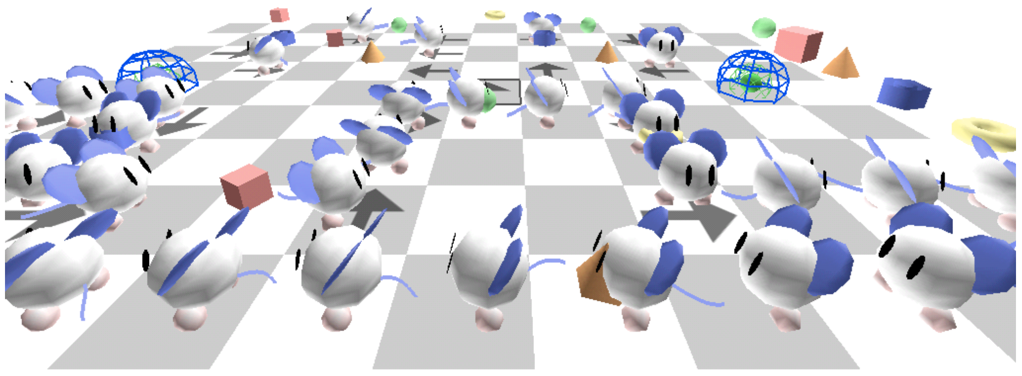


Figure 6.21: ChucK ChucK Rocket: from another viewpoint.

properties of the sounds. Eventually, a rhythmic pattern emerges (the striking) wherein the individual laptop orchestra players control both texture and synchronization."

Crystalis Composer: Ge Wang. This is a sonic rumination of crystal caves in the clouds, where the only sounds are those of the wind and the resonances of the crystals. It uses two simple instruments called the crystalis (based the Banded Waveguide synthesis technique [29]) and wind-o-lin. These instruments make use of the laptop keyboard (which controls pitch and resonance) and the trackpad (which the players "bow" in various patterns to generate sound). See instrument instructions (Figure 6.22).

TBA Composer: Ge Wang. On-the-fly programming, or live coding, is the practice of writing code in real-time to create music. This piece is our first attempt at large-scale, group live coding (15 humans/laptops) to create a single sound world. Players, divided into squadrons, follow instructions from a conducting live coder, who issues directives both in the form of code fragments (in the ChucK language)

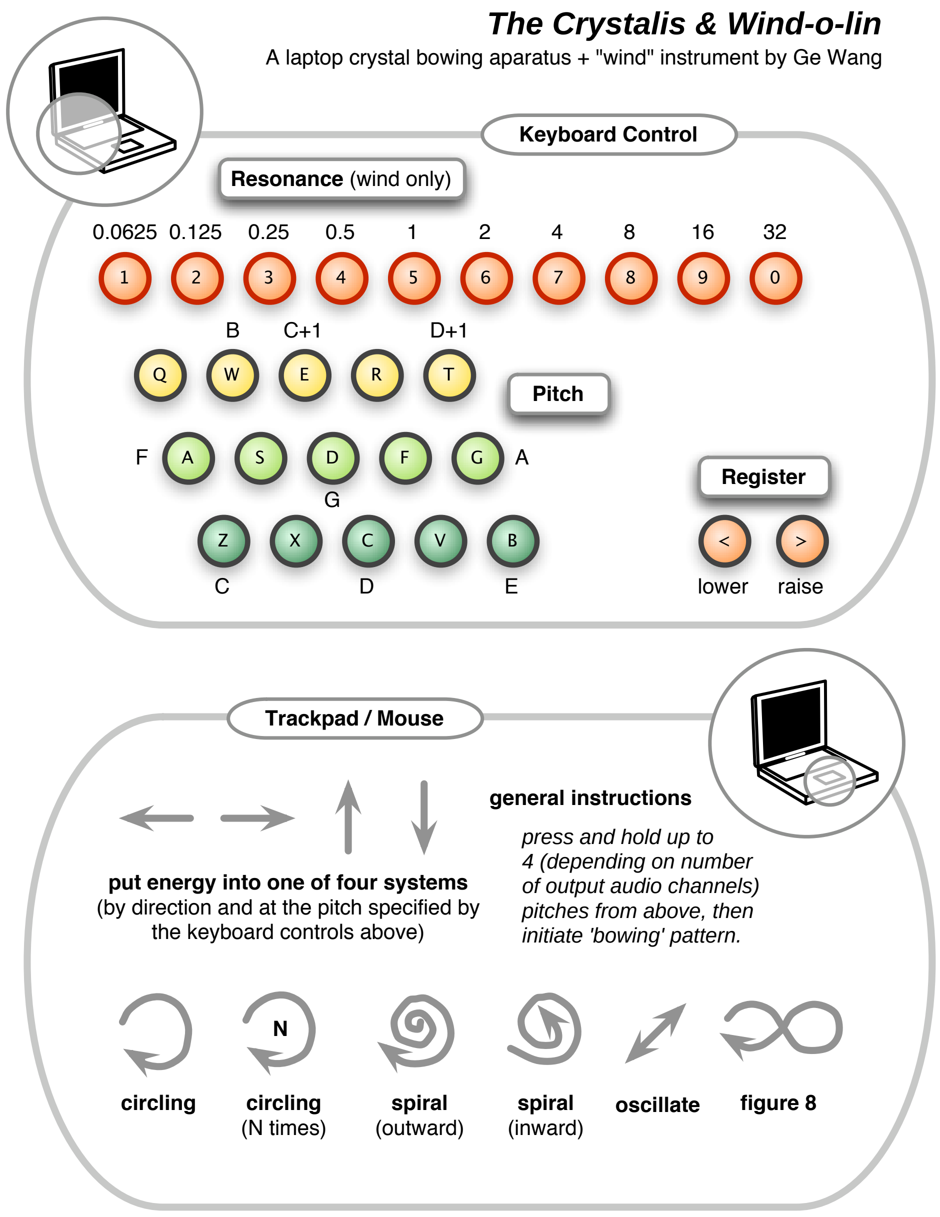
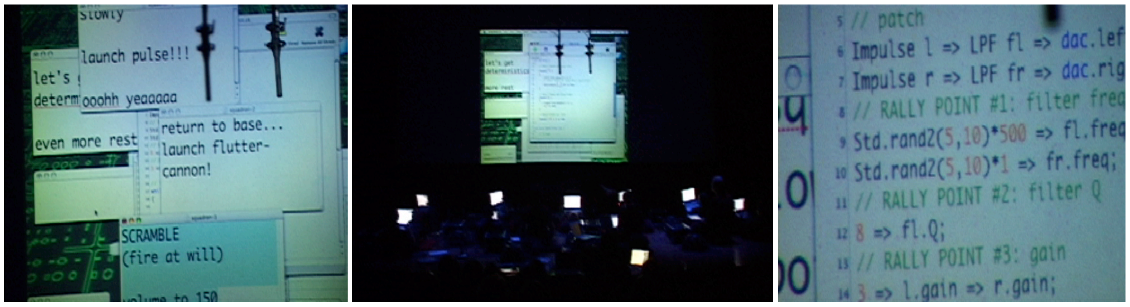

The Crystalis & Wind-o-lin
A laptop crystal bowing aparatus + "wind" instrument by Ge Wang
Keyboard Control
Resonance (wind only)
0.0625
0.125
0.25
0.5
1
2
4
8
16
32
1 2 3 4 5 6 7 8 9 0
B
C+1
D+1
Q W E R T
Pitch
F
A S D F G
A
G
Z X C V B
C
D
E
Register
<
>
lower
raise
Trackpad / Mouse
general instructions
put energy into one of four systems
(by direction and at the pitch specified by the keyboard controls above)
press and hold up to 4 (depending on number of output audio channels) pitches from above, then initiate 'bowing' pattern.
circling
N
circling (N times)
spiral (outward)
spiral (inward)
oscillate
figure 8

Figure 6.22: Crystalis: keyboard and trackpad mappings.



```
// patch
Impulse l => LPF fl => dac.lef
Impulse r => LPF fr => dac.rig
// RALLY POINT #1: filter freq
Std.rand2(5,10)*500 => fl.freq
Std.rand2(5,10)*1 => fr.freq;
// RALLY POINT #2: filter Q
8 => fl.Q;
// RALLY POINT #3: gain
3 => l.gain => r.gain;
```

Figure 6.23: TBA: orchestral live coding.

and sentence fragments (in the English language). In keeping with the crucial live coding tenet of revealing the process to the audience, the conducting machine will be projected 1) for all to observe and 2) as a means of instructing the ensemble.

Players begin with a simple code template (in the miniAudicle environment), which they modify over the course of the performance to create and sculpt sound. Operations include code modifications, adding code (+) to be rendered into sound, or replacing existing code (=) with updates. "Rally points" are set throughout the template to coordinate group coding bombardments. The piece alternates between detailed code changes and sections in which players are encouraged to improvise. In on-the-fly programming, the code is the instrument; and it is played via the act of programming. Also, we never really know what's going to happen next (expect glorious disasters). Until it is performed, the piece remains TBA to all, including the players (Figure 6.23).

PLOrk Beat Science Composers: Rebecca Fiebrink and Ge Wang. PLOrk Beat Science (PBS) is an electro-acoustic structured improvisation for 1 flute, 2 humans, 5 laptops, 5 pressure-sensitive finger drum pads, and 30 audio channels distributed among 5 hemispherical speakers. PBS was first created and performed in 2006 as a Princeton Laptop Orchestra (PLOrk) chamber piece, by Rebecca Fiebrink and Ge

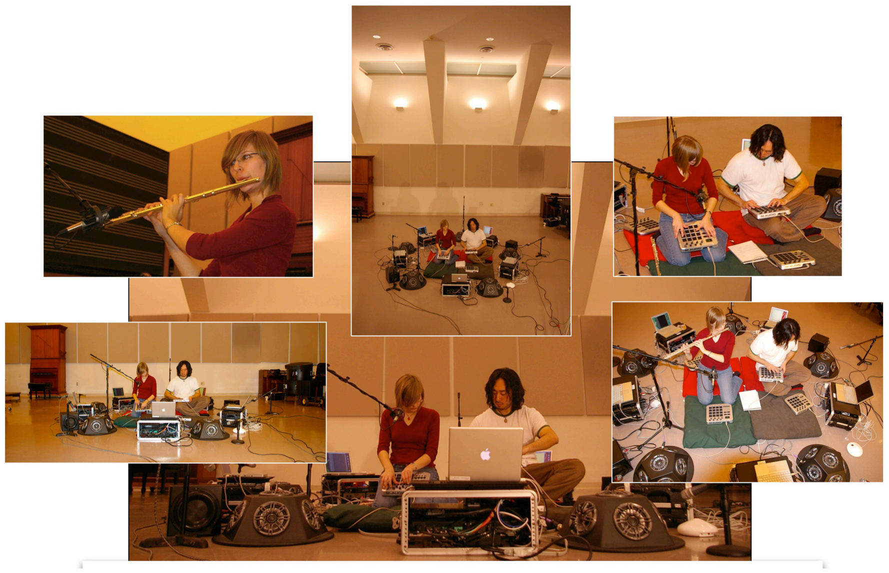


Figure 6.24: PLOrk Beat Science: Rebecca Fiebrink and Ge Wang.

Wang. Drawing from our work with PLOrk and taking inspiration (and 2/3 of our name) from Tabla Beat Science, PLOrk Beat Science reflects our interest in exploring new hybrid performances involving live acoustic instruments (flute, processed) in electronic chamber music settings (via laptops and hemispherical speakers, which radiate sound outwards from each localized instrument), creating crazy interactive beat machines, crafting new performance software and expressive controller mappings, and simply making music together (Figure 6.24, 6.25, 6.26).

Joy of Chant Composers: Rebecca Fiebrink, Ge Wang, and Perry Cook. A choir of simple (but glorious) singing synthesis models is controlled in real-time by players wielding joysticks and playing the laptop keyboard.

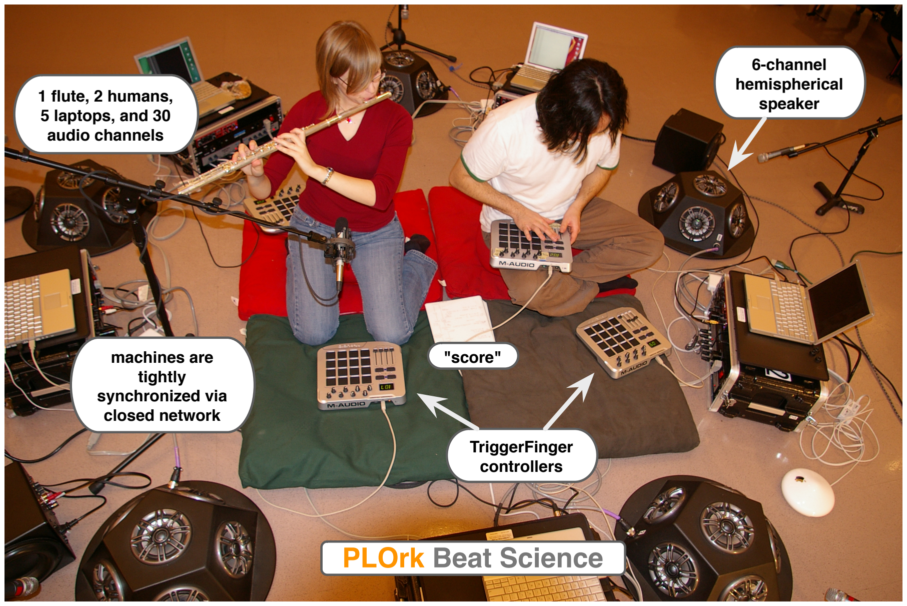

1 flute, 2 humans, 5 laptops, and 30 audio channels
6-channel hemispherical speaker
machines are tightly synchronized via closed network
"score"
TriggerFinger controllers
PLOrk Beat Science

Figure 6.25: PLOrk Beat Science: 1 flute, 2 humans, 5 laptops, 5 TriggerFingers, 30 audio channels.

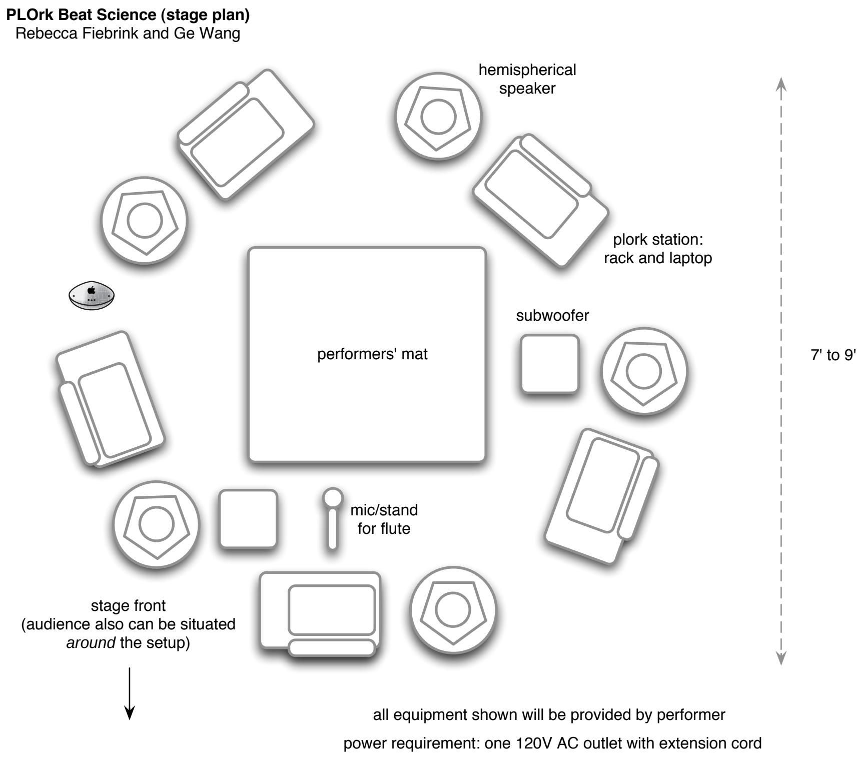

PLOrk Beat Science (stage plan)
Rebecca Fiebrink and Ge Wang
hemispherical speaker
plork station: rack and laptop
performers' mat
subwoofer
7' to 9'
mic/stand for flute
stage front (audience also can be situated *around* the setup)
all equipment shown will be provided by performer
power requirement: one 120V AC outlet with extension cord

Figure 6.26: PLOrk Beat Science: floor plan.

6.3.2 S.M.E.L.T.

The Small Musically Expressive Laptop Toolkit (S.M.E.L.T.) is an open-source toolkit to facilitate rapid development of and experimentation with expressive musical interfaces built on the laptop's native physical input capabilities (e.g., keyboard, mouse, sudden motion sensor, microphone) [31]. It's implemented in ChucK, and is based on work with the Princeton Laptop Orchestra.

The code is freely available (http://smelt.cs.princeton.edu/), and has served as the starting point for a large number of projects involving physical interaction, instrument building, and sound mapping. It also serves as a useful resource for teaching in laptop orchestras.

6.3.3 ChucK for TAPESTREA

TAPESTREA (or *taps*) is a unified framework for interactively analyzing, transforming and synthesizing complex sounds [62] - being developed at Princeton University, led by Ananya Misra. Given one or more recordings, it provides well-defined means to: 1) identify points of interest in the sound and extract them into reusable template, 2) transform sound components independently of the background and/or other events, 3) continually resynthesize the background texture in a perceptually convincing manner, 4) controllably place event templates over backgrounds, using a novel graphical user interface and/or scripts written in the ChucK language, and 5) leverage similarity based retrieval to locate other interesting sound components. *Taps* provides a new way to completely transform a sound scene, dynamically generate soundscapes of unlimited length, and compose and design sound by combining elements from different recordings.

Currently, *taps* provides a TAPESTREA-specific API for ChucK, in addition to making the full language available to the user. In this case, ChucK presents the ability to describe temporally precise and concurrent control over the transformation and synthesis. For example, TAPESTREA uses Spectral Modeling Synthesis [76] to model *deterministic* components of a sound, and then allows the user to then control parameters such as frequency warping, time stretching, and event density to greatly transform the sound templates. Using ChucK, one can script, to a highly precise degree, how these parameters might vary over time. The API couples these parameter values to graphical user interface elements (e.g., sliders), allowing code to control multiple elements accurately and concurrently, and in tandem with human interaction, making it amenable for experimentation, composition, teaching, and perhaps even live performance.

6.4 Additional and Potential Applications

Here we wrap up by describing some ongoing as well as planned applications using ChucK. Potential future directions for the ChucK programming language itself are discussed in the *Conclusions* chapter.

Writing "White-box" Unit Generators/Analyzers. Since the ChucK programmer is able to talk about time in a globally consistent and arbitrarily fine level, it is possible to construct unit generators and analyzers directly in ChucK. This greatly reduces the need to go outside the language when implementing, refining, and expanding low-level functionality. This may be especially beneficial for working with algorithms that are well-known in general, but have a large degree of low-level variability (these include Linear Predictive Coding [64], granular synthesis [71], For-mat Wave Functions (FOF's) [73], and other classes of audio algorithms [72, 19]). The determinism and precision provided by ChucK can also benefit research in new algorithms, both in terms being able to express arbitrarily complex and low-level timing, and by provide a rapid-protoptyping environment for experimentation (recall our goal to "hide the mundane, expose true control"). Furthermore, writing unit generators "in-language" can yield new unit modules that are immediately ready for "open-source" distribution and exchange.

Musical Robots and Affective Computing. ChucK is currently being used for research in musically intelligent musical robots as well as in the related area of affective computing [41, 43, 42], where machines endeavor to discern emotion and musical meaning associated via various sensory modalities (e.g., audio, vision, motion tracking). Figure 6.27 shows Ajay Kapur playing the e-sitar with Mahadevibot, a musical drumming robot.

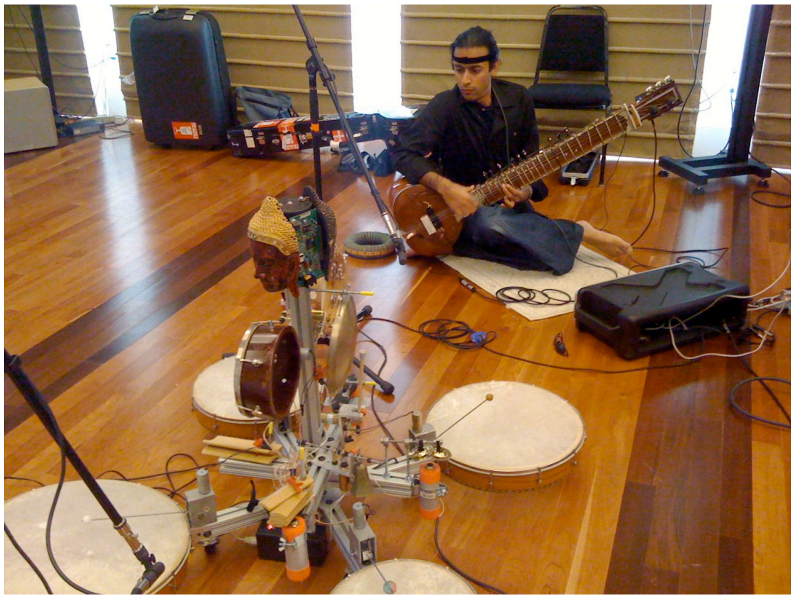


Figure 6.27: Mahadevibot: musical robotic, playing with a human performer; various software components implemented in ChucK.

Networked Performance is concerned with the technological, aesthetic, and musical aspects of distributive, live, and high-quality audio over networks [14, 45, 44] (Figure 6.28). This is a growing area of research and it would be interesting to explore ChucK as an additional tool for implementing both underlying systems for networked audio, as well as to build "client-side" networked musical instruments.

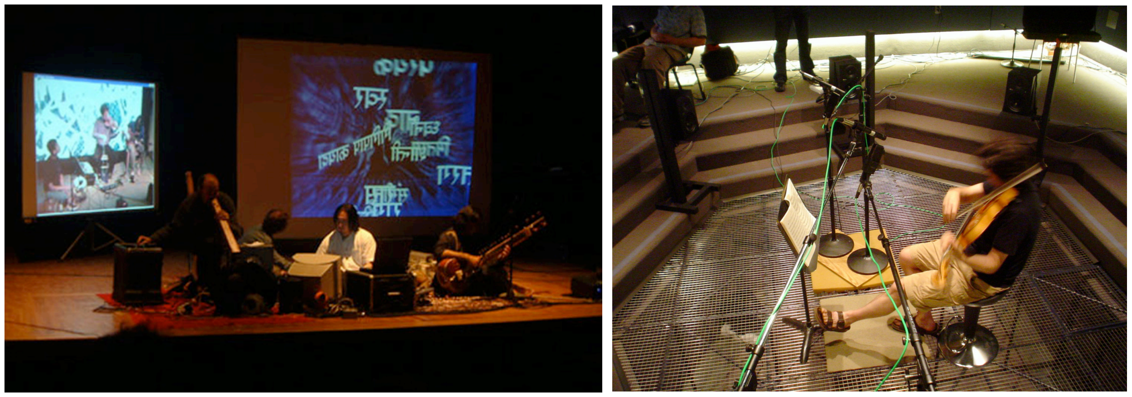


Figure 6.28: Networked Audio Performances: Gigapop Ritual (2003, left) between McGill University and Princeton University; right: performance between CCRMA and Banff with a distributed St. Lawrence String Quartet using JackTrip. Both systems used C/C++ based software, though networked audio may be a potential application of ChucK in the future.

Audio Mosaicing can be described as the concatenation of segment of sounds, whereby the segments are choosing by some metric (e.g., distance in feature space) in relation to a input sound stream. This can interpreted as a data-driven granular synthesis. The MoSievius project [47] explores using ChucK as the sonic engine, both for feature extraction and low-level synthesis.

Feature-based Synthesis (FBS) is a promising technique in development for sonic modeling, representation, compression, as well as an evaluation tool for a variety of Music Information Retrieval tasks [37]. Given a set of feature values and a choice of synthesis algorithm, FBS seeks to find parameters for the synthesis algorithm to manifest those feature values, through various learning and searching subsystems. The currently implementation of FBS exists as a C++ library. It'd be interesting to explore the usage of ChucK to aid in rapid prototyping of various parts of the system, and perhaps to implement new components.

Transmission and Archival of Audio Algorithms. Since ChucK has no dependencies on system timing, audio synthesis is guaranteed to be computed in a deterministic way. A consistent and precise notion of time combined with explicit readability allow ChucK to precisely specify synthesis algorithms, for education and for archiving.

Chapter 7

Conclusion

7.1 Contributions

The contributions of this work include the following.

- **A time-based programming mechanism for synchronous, ultra-precise audio synthesis and real-time audio analysis**. ChucK's timing mechanism unifies time across an immense range of granularities, using the same model to drive ultra-precise DSP-level processes, as well as interactive control interactions, to higher level sonic and musical structures, to still higher processes dealing with timing at the order of days, weeks, and even years.
- **A non-preemptive, time/event-based concurrent programming model** provides fundamental flexibility and readability without incurring many of the difficulties of programming parallelism. This allows expressive representation and specification of parallelism, and is amenable for teaching concurrency in introductory computer music/programming courses, and is effective for use by expert programmers.

- **A ChucKian approach to writing coding and designing audio programs on-the-fly.** This rapid prototyping mentality has wide ramifications in the way we think about coding audio, in designing/testing software (particular for real-time audio), as well as new paradigms and practices in computer-mediated live performance.
- **Extended case studies of using, teaching, composing, and performing with ChucK**. These show the power of teaching programming via music, and vice versa - and how these two disciplines can reinforce each other, making learning fun and perhaps "by-accident" (and thus nearly inevitable). These ideas are deeply embodied in the instantiation of the Princeton and Stanford Laptop Orchestras.

Overall, a contribution is the "holistic" sum of a set of ideas, approaches, and a programming language platform to serve a diverse user community.

7.1.1 The Meaning of "Strongly-timed"

A *strongly-timed* programming language, as we've defined it, is one in which there is a well-defined separation of synchronous logical time from real-time. Via this separation, one can more easily specify, debug, and reason about the programs written in the language, allowing programs to be designed and specified without having to worry about external factors, such as machine speed, portability and timing behavior across different systems. Furthermore, this mechanism can be used to specify powerful deterministic concurrency. Due to this model, ChucK appears to be unique in supporting tightly interleaved control and audio computation, allowing the programmer to move rather smoothly from the domain of digital signal processing at the sample level to the more gestural levels of control.

7.2 Future Work

In this penultimate section, we explore and discuss some potential future directions for the ChucK programming language, environment, and intrinsic applications. Some of these are already underway at the time of this writing. All of these await new research and (we believe) hold good potentials for continuing to provide new ways of thinking about audio, music, and programming.

7.2.1 Exploring Analysis, Rapid Prototyping, Learning

At the time of this writing, we've established only a small part of the groundwork for specifying audio analysis in ChucK, and are continuing to investigate new possibilities. Recent years have seen a proliferation of general tools and frameworks to perform audio analysis, particularly in the area of Music Information Retrieval (MIR). Commonly used tools specialized for MIR include MARSYAS [88], CLAM [2], SndObj [48], MATLAB/Octave [56], M2K [28], and jAudio [16] / jMIR [17]. These tools support feature extraction from audio files, and classification and learning from these features.

These are primarily libraries and frameworks; as such, they offer programmability at a different level than languages such as ChucK. As far as we can tell, there is no high-level language specialized for analysis tasks, much less one focused on support for integrated real-time analysis and synthesis. Therefore, we feel that a ChucK-oriented programming model can be potentially interesting in its own right, and may serve as a complimentary tool to existing systems.

Rapid prototyping tools have an established role in MIR. M2K, for example, has been developed for this purpose. It provides a graphical patching environment for feature extractors, classifiers, and other modules. Dataflow and functionality in M2K itineraries are determined by connections between built-in and user-created objects, implemented in Java. MARSYAS also provides support for rapid prototyping, via Python and MARSYAS Scripting Language [3]. Our approach differs from these in that we hope to enable rapid experimentation from a single, unified high-level programming platform with low-level control, further reducing turnaround time and itself enabling (and perhaps even suggesting) new algorithms and applications.

The motivations behind creating such a language framework include enabling more people to experiment with, prototype, and create new MIR algorithms and systems. Using this combined analysis/synthesis framework, programmers should be able to rapidly prototype and implement analysis and synthesis tasks – and even do so on-the-fly. Like the synthesis framework, the analysis/synthesis code should represent the underlying algorithms and dataflow precisely and clearly. Overall, we hope to present language solution that meets these criteria and that can be equally suitable for audio research (e.g., synthesis, spectral processing, feature extraction, machine learning, music information retrieval), pedagogy, composition, and musical performance.

A vast array of potential current and future work remains to be explored, including new language syntax and semantics for specifying audio analysis, language and library support for performing feature extraction as well as state-of-the-art machine learning algorithms, to things we can't yet predict. Some overarching goals include 1) providing an expressive MIR rapid prototyping workbench for use in research and teaching, and 2) exploring the tight coupling of MIR, synthesis, live performance. In keeping with the philosophical motivations of this thesis, we hope to encourage different ways to think about programming for analysis and synthesis.

7.2.2 Worlds for Collaborative Social Audio Programming

Another exciting future area of research might be to investigate the social aspects of *collaborative social audio programming*. It seems natural to leverage the strongly-timed, concurrent aspects of ChucK, combined with on-the-fly programming, and the unique social interaction of computer-simulated worlds. The idea is to create a collaborative, *massively multi-user interaction space* based around the ChucK language (possibly extending the Audicle into a collaborative *Co-Audicle*).

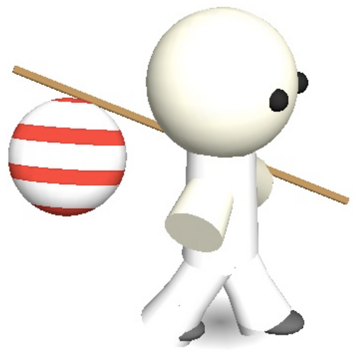


Figure 7.1: Future work: a denizen in the envisioned collaborative social audio programming virtual world.

On-the-fly programming views code as an expressive musical instrument and the act of programming as performance. This research direction, combined with today's fast networks and existing research in audio over networks [14, 44, 5], provides a unique opportunity to empower many composers, audio researchers, instructors, students, and musicians to interact in a common audio programming virtual world (Figures 7.1 and 7.2). In such a realm, the exchange of ideas can take place via code, graphics, text, live audio, and perhaps much more. We believe this has the potential to enhance and transform how we make music as a group, conduct audio research, and carry out computer music pedagogy. (Also, it might be a great deal of fun.)

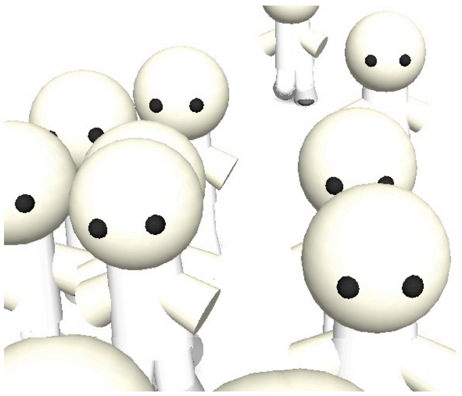


Figure 7.2: Future work: many entities collaboratively live coding in same virtual space.

7.2.3 Planned Language Features

Optimization

As noted in Chapter 3, priority is given to try to maximize flexibility and readability first of all, and designing for high performance and throughput only when it doesn't conflict with the former (subscribing to the rule of thumb suggested to the author by his college CS professor Owen Astrachan: "Make it work, make it right, make it fast, and make it small – in that order"). Truly, the rapid advancements in computing power have afforded this tradeoff, and in many ways, we are still trying to make parts of the language work (and hoping to get some things right in the process). At the same time, we have been investigating various potential optimizations that would offer significant higher throughput without sacrificing the flexibility and precision of the language.

One such potential optimization is *adaptive block processing* of audio samples. Currently, the *sample-at-a-time* audio synthesis network provides immense control over time and granularity, but incurs large performance overheads from repeated function calls as well as being difficult in taking advantage of vectorization and processor pipelining. One potential solution to this issue to implement adaptive block processing, whereby the synthesis/analysis engine works closely with the ChucK shreduler and perform audio computations in block sizes that are appropriate with the current time-based shreduling. Two issues will need to be addressed in such an architecture. One is detecting cycles in the audio network,as the UGen's in the cycle may still need to compute one sample at the time for feedback (though presence of delay elements may relax this requirement). The second issue is to define a "maximum block size" to allow existing "best effort" asynchronous input to be processed (e.g., MIDI, HID, and OSC).

Taking advantage of Multi-core Processors

At the time of this writing (2006-2008), a focus in computer architecture design is that of shifting from increasing clock speed on a single CPU to multiple symmetrical cores, moving to single machines with hundreds or perhaps many more cores. The current architecture of ChucK places all audio and language computations on the same kernel thread, managing the shred-based user-level concurrency entirely in the ChucK virtual machine, making it difficult to take advantage of many cores. This is an issue that applies to most current computer music programming systems (as well as many software systems in general), and likely new underlying architectures (or even end-programmer syntax and semantics) will need to address this issue in ChucK. One perspective on the problem is that currently the computing community at large hasn't yet produced an agreed-upon, generalized lower-level programming paradigm/language (e.g., at the C/C++ level) that can flexibly specify synchronous, multi-core friendly programs. At the ChucK system level, interesting questions likely lie at the intersection of concurrent low-level implementation and the semantics of the ChucK language. In any case, this remains an open issue that has great ramifications on the future of computing and programming for music, in ChucK and more broadly.

Graphics

Another area well worth investigating is graphics programming integrated with audio in ChucK. To be able to specify real-time graphics, image, and even video processing in the same strongly-timed framework for audio can potentially lead to great possibilities. This combined approach has long been demonstrated to be compelling [50, 70]. The current plan is to both investigate more tightly-coupled communications between ChucK and graphical programming environments such as Processing, as well as providing "in-language" support and API for sound-coupled graphics.

7.2.4 Laptop Orchestras

Since the beginning, ChucK (for better or for worse) has served as a primary software platform for teaching, programming, instrument building, and live performance in the Princeton Laptop Orchestra and later in the Stanford Laptop Orchestra (Figure 7.3). As researchers in the emerging medium of the laptop orchestra, we hope to continue investigating its musical, sonic, and pedagogically potential and help others who are interested in doing the same. In Dan Trueman's dissertation [84], he postulated the notion of making *electronic chamber music*, directly prognosticating and leading to the instantiation of the laptop orchestra. It is the hope of the author of this thesis, and other laptop orchestra progenitors, that the laptop orchestra will proliferate as a sustained artistic medium, providing an established (and yet open) platform for music-making. We look forward to intensely investigating how ChucK might be better used in these contexts, possibly combined with aforementioned ideas such as collaborative social programming and audio over networks. At the same time, may ChucK continue to be one of many potential tools in the software palette of the laptop orchestra.

Figure 7.3: Laptop Orchestras: PLOrk, SLOrk – and hopefully beyond!

7.2.5 ChucK and the Mobile Phone

In recent years, the mobile phone has firmly established itself as the single piece of technology that transcends nearly every cultural, social, and economical barrier [30]. It's perhaps paradoxical that the modern mobile phone are simply smaller computers in every sense, and yet it presents a fundamentally different set of interactive, sonic, social, and technological parameters. It would be incredibly exciting to continue to explore its possibilities. In early 2008, the first mobile phone orchestra (that we are aware of), MoPhO, was founded at Stanford University's CCRMA [100], though not yet using ChucK on the phones. The next steps might be to investigate ChucK as well as audio programming in general on and for mobile platforms. Like the laptop orchestra, this is still nascent (even more so), and holds great potential to transform music-making.

7.3 Concluding Remarks

In this thesis, we presented the ChucK programming language, its ideas, design goals, language specifications, implementation, and the various paradigms and ways of thinking associated with the language. Additionally, we examined several core applications of the language, and evaluated ChucK as a programming tool as well as a pedagogical vehicle. While much has been investigated, perhaps even more remains to be discovered and explored. It is the hope of the author, the ChucK development team, and the ChucK community at large, that this investigation continues, remembering that while technology is central to what we do, it is what we do with technology that truly matters.

Thank you for reading.

Bibliography

[1] ADA. *The Programming Language ADA Reference Manual*. Springer-Verlag, LNCS 155, 1983.

[2] Xavier Amatrian, Pau Arumi, and Miguel Ramirez. CLAM, Yet Another Library For Audio and Music Processing. In *ACM Conference on Object-Oriented Programming*, 2002.

[3] David Anderson and Ron Kuivila. Formula: A Programming Language for Expressive Computer Music. *IEEE Computer*, 24(7):12–21, 1991.

[4] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[5] Alvaro Barbosa. *Computer-Supported Cooperative Work for Music Applications*. PhD thesis, Music Technology Group, Pompeu Fabra University, 2006.

[6] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] Richard Boulanger. *The Csound Book*. MIT Press, 2000.

[8] Eli Brandt. Temporal Type Constructors for Computer Music Programming. In *International Computer Music Conference*, 2000.

[9] Eli Brandt. Implementing Temporal Type Constructors for Music Programming. In *International Computer Music Conference*, 2001.

[10] Andrew Brown and Andrew Sorensen. Dynamic Media Arts Programming in Impromptu. In *ACM SIGCHI Conference on Creativity and Cognition*, 2007.

[11] Phil Burk. JSyn - A Real-time Synthesis API for Java. In *International Computer Music Conference*, 1998.

[12] Phil Burk, Larry Polansky, and David Rosenboom. HSML: A Theoretical Overview. *Perspectives of New Music*, 28(2), 1990.

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Annual Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[14] Chris Chafe, Scott Wilson, Randall Leitstikow, Dave Chisholm, and Gary Scavone. A Simplified Approach to High Quality Music and Sound Over IP. In *Digital Audio Effects*, 2000.

[15] John Chowning. Keynote: International Computer Music Conference 2007. Keynote Address, August 2007.

[16] Pierre Cointe and Xavier Rodet. Formes: An Object and Time Oriented System for Music Composition and Synthesis. In *ACM Symposium on LISP and Functional Programming*, 1984.

[17] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live Coding Techniques for Laptop Performance. *Organised Sound*, 8(3):321–330, 2003.

[18] Perry R. Cook. *Identification of Control Parameters in an Articulatory Vocal Tract Model with Applications to the Synthesis of Singing.* PhD thesis, Stanford University, 1991.

[19] Perry R. Cook. *Real Sound Synthesis for Interative Applications.* A. K. Peters, 2002.

[20] Perry R. Cook and Gary Scavone. The Synthesis Toolkit (STK). In *International Computer Music Conference*, Beijing, China, 1999.

[21] Roger Dannenberg. A Real-Time Scheduler / Dispatcher. In *International Computer Music Conference*, 1988.

[22] Roger Dannenberg. The Canon Score Language. *Computer Music Journal*, 13(1):47–56, 1989.

[23] Roger Dannenberg. Abstract Time Warping of Compound Events and Signals. *Computer Music Journal*, 21(3):61–70, 1997.

[24] Roger Dannenberg. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal*, 21(3):50–60, 1997.

[25] Roger Dannenberg and Eli Brandt. A Flexible Real-Time Software Synthesis System. In *International Computer Music Conference*, 1996.

[26] Roger Dannenberg, P. McAvinney, and D. Rubine. Arctic: A Functional Language for Real-Time Control. *Computer Music Journal*, 10(4):67–78, 1986.

[27] F. Dechelle, R. Borghesi, M. Ceccco, E. Maggi, B. Rovan, and N. Schnell. jMax: A New Java-based Editing and Control System for Real-time Musical Applications. *Computer Music Journal*, 23(3):50–58, 1998.

[28] Stephen Downie. International Music Information Retrieval Evaluation Laboratory (IMIRSEL): Introducing M2K and D2K. Handout at International Conference on Music Information Retrieval, 2004.

[29] Georg Essl, Stefania Serafin, Perry R. Cook, and Julius O. Smith. Musicial Applications of Banded Waveguides. *Computer Music Journal*, 18(1):51–63, 2004.

[30] Georg Essl, Ge Wang, and Michael Rohs. Developements and Challenges of Turning Mobile Phones in Generic Music Performance Platforms. In *Mobile Music Workshop*, Vienna, Austria, 2008.

[31] Rebecca Fiebrink, Ge Wang, and Perry R. Cook. Don't Forget the Laptop: Using Native Input Capabilities for Expressive Musical Control. In *International Conference on New Interfaces for Musical Expression*, pages 164–167, New York City, U.S.A., 2007.

[32] Brad Garton and David Topper. RTcmix Online Documentation. http://rtcmix.org/.

[33] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. SIGNAL: A Data Flow Oriented Language for Signal Processing. IRISA Report 246, IRISA, Rennes, France, 1985.

[34] Nicholas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[35] David Harel and Amir Pnueli. On the Developement of Reactive Systems: Logic and Models of Concurrent Systems. In *NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, pages 54–64, 1985.

[36] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[37] Matt Hoffman and Perry R. Cook. Feature-based Synthesis: A Tool for Evaluating, Designing, and Interacting with Music IR Systems. In *International Conference on Music Information Retrieval*, 2006.

[38] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore Music Notation - An Algebra of Music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[39] David A. Jaffe and Julius O. Smith. Extensions of the Karplus-Strong Plucked String Algorithm. *Computer Music Journal*, 7(2):56–69, 1983.

[40] Kevin Kaplus and Alex Strong. Digital Synthesis of Plucked String and Drum Timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[41] Ajay Kapur. A History of Robotic Musical Instruments. In *International Computer Music Conference*, 2005.

[42] Ajay Kapur, Adam R. Tindale, Manjinder S. Benning, and Peter F. Driessen. The *KiOm*: A Paradigm for Collaborative Controller Design. In *International Computer Music Conference*, 2006.

[43] Ajay Kapur, George Tzanetakis, Naznin Virji-Babul, Ge Wang, and Perry R. Cook. A Framework for Sonification of VICON Motion Capture Data. In *International Conference on Digital Audio Effects*, Madrid, Spain, 2005.

[44] Ajay Kapur, Ge Wang, Philip Davidson, and Perry R. Cook. Interactive Network Media: A Dream Worth Dreaming? *Organised Sound*, 10(3):209–219, 2005.

[45] Ajay Kapur, Ge Wang, Philip Davidson, Perry R. Cook, Dan Trueman, Tae Hong Park, and Manjul Bhargava. The Gigapop Ritual: A Live Networked Performance for Two Electronic Dholaks, Digital Spoon, DigitalDoo, 6 String Electric Violin, Rbow, Sitar, Tabla, and Bass Guitar. In *International Conference on New Interfaces for Musical Expression*, Montreal, Canada, 2003.

[46] Paul Lansky. *CMIX Program Documentation.* Princeton, NJ, U.S.A., 1987.

[47] Ari Lazier and Perry R. Cook. MoSievius: Feature-based Interactive Audio Mosaicing. In *International Conference on Digital Audio Effects*, London, UK, 2003.

[48] Victor Lazzarini. The Sound Object Library. *Organised Sound*, 5(1):35–49, 2000.

[49] Edward A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, U. C. Berkeley, 2003.

[50] Golan Levin. *Painterly Interfaces for Audiovisual Performance*. Master's thesis, Massachusetts Insititute of Technology, 1994.

[51] George Lewis. Too Many Notes: Computers, Complexity and Culture in Voyager. *Leonardo Music Journal*, 10:33–39, 2000.

[52] Gareth Loy. The CARL System: Premises, History, and Fate. *Computer Music Journal*, 26(4):52–60, 2002.

[53] Wacom Co. Ltd. Wacom. http://www.wacom.com/.

[54] Max Mathews. *The Technology of Computer Music*. MIT Press, 1969.

[55] Max Mathews. Keynote: International Computer Music Conference 2006. Keynote Address, November 2006.

[56] Mathworks. *MATLAB Documentation*.

[57] Dominic Mazzoni and Roger Dannenberg. Audacity: Free Audio Editor and Recorder. http://audacity.sourceforge.net/, 2008.

[58] James McCartney. SuperCollider: A New Real-time Synthesis Language. In *International Computer Music Conference*, 1996.

[59] James McCartney. Rethinking the Computer Music Programming Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[60] Alex McLean. Hacking Perl in Nightclubs. O'Reily Media Inc. http://www.perl.com/pub/a/2004/08/31/livecode.html, 2004.

[61] Ananya Misra, Ge Wang, and Perry R. Cook. SndTools: Real-time Audio DSP and 3D Visualization. In *International Computer Music Conference*, Barcelona, Spain, 2005.

[62] Ananya Misra, Ge Wang, and Perry R. Cook. Musical Tapestry: Re-composing Natural Sounds. *Journal of New Music Research*, 2008.

[63] F. Richard Moore. The Computer Audio Research Laboratory at UCSD. *Computer Music Journal*, 6(1):18–29, 1982.

[64] James A. Moorer. The Use of Linear Prediction of Speech in Computer Music Applications. *Journal of the Audio Engineering Society*, 27(3):134–140, 1979.

[65] Alan Oppenheim, Alan Willsky, and S Hamid Nawab. *Signals and Systems*. Prentice Hall, 2002.

[66] Randy Pausch, Tommy Burnett, A. C. Capeheart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Kogai, and Jeff White. Alice: Rapid Prototyping System for Virtual Reality. *IEEE Computer Graphics and Applications*, 1995.

[67] Stephen T. Pope. Machine Tongues XV: Three Packages for Software Sound Synthesis. *Computer Music Journal*, 17(2):23–54, 1993.

[68] Miller Puckett. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[69] Miller Puckett. Pure Data. In *International Computer Music Conference*, 1996.

[70] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[71] Curtis Roads. *Foundations of Computer Music*, chapter Granular Synthesis of Sound. MIT Press, 1985.

[72] Curtis Roads. *Computer Music Tutorial.* MIT Press, 1999.

[73] Xavier Rodet, Yves Potard, and Jean-Baptiste Barriere. The CHANT Project: From the Synthesis of the Singing Voice to Synthesis in General. *Computer Music Journal*, 8(3):15–31, 1984.

[74] Spencer Salazar, Ge Wang, and Perry R. Cook. miniAudicle and ChucK Shell: New Interfaces for ChucK Development and Performance. In *International Computer Music Conference*, New Orleans, U.S.A., 2006.

[75] Bill Schottstaedt. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):3–37, 1994.

[76] Xavier Serra and Julius O. Smith. Spectral Modeling Synthesis. In *International Computer Music Conference*, 1989.

[77] Chris Shaw, Jiandong Liang, Mark Green, and Yunqi Sun. The Decoupled Simulation Model for Virtual Reality System. In *ACM SIGCHI Human Factors in Computer Systems Conference*, 1992.

[78] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 2006.

[79] Scott Smallwood, Dan Trueman, Perry R. Cook, and Ge Wang. Composing for Laptop Orchestra. *Computer Music Journal*, 32(1):9–25, 2008.

[80] Julius O. Smith. Online Publications. http://ccrma.stanford.edu/˜jos/.

[81] Ken Steiglitz. *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music.* Prentice Hall, 1996.

[82] TOPLAP. Temporary Organization for the Proliferation of Live Audio Programming, 2004.

[83] David S. Touretzky. *LISP: A Gental Introduction to Symbolic Computation*. Harper and Row, 1984.

[84] Dan Trueman. *Reinventing the Violin*. PhD thesis, Princeton University, 1999.

[85] Dan Trueman. Why a Laptop Orchestra? *Organised Sound*, 12(2):171–179, 2007.

[86] Dan Trueman, Perry R. Cook, Scott Smallwood, and Ge Wang. PLOrk: Princeton Laptop Orchestra, Year 1. In *International Computer Music Conference*, New Orleans, U.S.A., 2006.

[87] Alan Turing. On Computer Numbers, with an Application to the Entscheidungsproblem. In *London Mathematical Society*, pages 230–265, 1936.

[88] George Tzanetakis and Perry R. Cook. MARSYAS: A Framework for Audio Analysis. *Organised Sound*, 4(3):169–175, 2000.

[89] Barry Vercoe. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. MIT Media Lab.

[90] Barry Vercoe. *Reference Manual for the MUSIC 11 Sound Synthesis Language*. MIT Experimental Music Studio.

[91] Barry Vercoe and Dan Ellis. Real-Time Csound: Software Synthesis with Sensing and Control. In *International Workshop on Memory Management*, 1990.

[92] Ge Wang. *The ChucK Language Specification*, online http://chuck.cs.princeton.edu/doc/language/, 2004-2008.

[93] Ge Wang. *Cambridge Companion to Electronic Music*, chapter 5: A History of Music and Programming. Cambridge University Press, 2008.

[94] Ge Wang. *Stanford Laptop Orchestra (SLOrk)*, homepage http://slork.stanford.edu/, 2008.

[95] Ge Wang and Perry R. Cook. ChucK: A Concurrent, On-the-fly Audio Programming Language. In *International Computer Music Conference*, Singapore, 2003.

[96] Ge Wang and Perry R. Cook. ChucK: A Programming Language for On-the-fly, Real-time Audio Synthesis and Multimedia. In *ACM Multimedia*, New York City, U.S.A., 2004.

[97] Ge Wang and Perry R. Cook. On-the-fly Programming: Using Code as an Expressive Musical Instrument. In *International Conference on New Interfaces for Musical Expression*, Hamamatsu, Japan, 2004.

[98] Ge Wang and Perry R. Cook. The Audicle: A Context-sensitive, On-the-fly Audio Programming Environ/mentality. In *International Computer Music Conference*, Miami, U.S.A., 2004.

[99] Ge Wang, Perry R. Cook, and Ananya Misra. Designing and Implementing the ChucK Programming Language. In *International Computer Music Conference*, Barcelona, Spain, 2005.

[100] Ge Wang, Georg Essl, and Henri Pentinnen. MoPhO: Do Mobile Phones Dream of Electric Orchestras? In *International Computer Music Conference*, Belfast, Ireland, 2008.

[101] Ge Wang, Rebecca Fiebrink, and Perry R. Cook. Combining Analysis and Synthesis in the ChucK Programming Language. In *International Computer Music Conference*, Copenhagen, Denmark, 2007.

[102] Ge Wang, Ananya Misra, Ajay Kapur, and Perry R. Cook. Yeah ChucK It! => Dynamic, Controllable Interface Mapping. In *International Conference on New Interfaces for Musical Expression*, Vancouver, BC, 2005.

[103] Ge Wang, Dan Trueman, Scott Smallwood, and Perry R. Cook. The Laptop Orchestra as Classroom. *Computer Music Journal*, 32(1):26–37, 2008.

[104] Matt Wright and Adrian Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *International Computer Music Conference*, 1997.

[105] Zeitwissen. Elektronische Musik: Tanz den Maschinencode. *Zeitwissen*, page 96, 2006.

‘