LogoRhythms: Introductory Audio Programming for Computer Musicians in a Functional Language Paradigm

Aaron Hechmer[1], Adam Tindale[2], George Tzanetakis[3]

Abstract - **Teaching computer music presents opportunities and challenges at both secondary and university levels by bringing together students with widely varying exposures to and interests for mathematics and computer programming. Visual languages like MAX/MSP are popular with many musicians, but the idiom doesn't necessarily transfer well to a text language such as Java or C++, languages that might be used in a wider variety of programming problems. Our design challenge with LogoRhythms was to create a forgiving text based API that allows the neophyte programmer to explore programming and low-level digital audio manipulations. Since any musical composition is essentially a novel program, the opportunity for custom software is endless and the programming task given as a creative endeavor. LogoRhythms encourages functional style programming. Examples are provided showing lists and higher order functions used to create simple harmonies and melodies with a discussion of how to balance abstracting elegance with "abstracting elusiveness."**

Index Terms - Logo, Audio, Programming Languages, Music, Computer Music, Computer Literacy.

INTRODUCTION

The University of Victoria, BC has recently begun to offer a joint undergraduate degree program between the Computer Science and Music Departments. The program brings together a diverse group of students whose technical approach, comfort and expertise with hardware, software and mathematics varies widely. The students remain unified in an interest in creating, performing and analyzing music. Similarly, their musical backgrounds draw from many sources: improvisational, play-by-ear, dj's and classically trained performers are all represented. Almost all the students come, in some way, to think of sound mathematically, if not formulaically then graphically with qualitative descriptions of concepts like phase, filtering or spectrum. Beyond simply becoming virtuoso performers or competent composers, these computer musicians usually take on the role of luthier, building their own instruments from a wide array of hardware and software components. One way to categorize the software most frequently used by computer musicians is into three bins: graphical applications such as sound editors that often build their interfaces around an oscilloscope window giving waveform or spectrum, visual programming languages where functions are represented as graphical objects with pipes connecting them and, finally, traditional high-level typed-text languages.

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

[1] Aaron Hechmer, Computer Science Department, University of Victoria, BC ahechmer@uvic.ca
[2] Adam Tindale, Computer Science , ECE and Music Departments, University of Victoria, BC
[3] George Tzanetakis, Computer Science and Music Departments, University of Victoria, BC gtzan@cs.uvic.ca

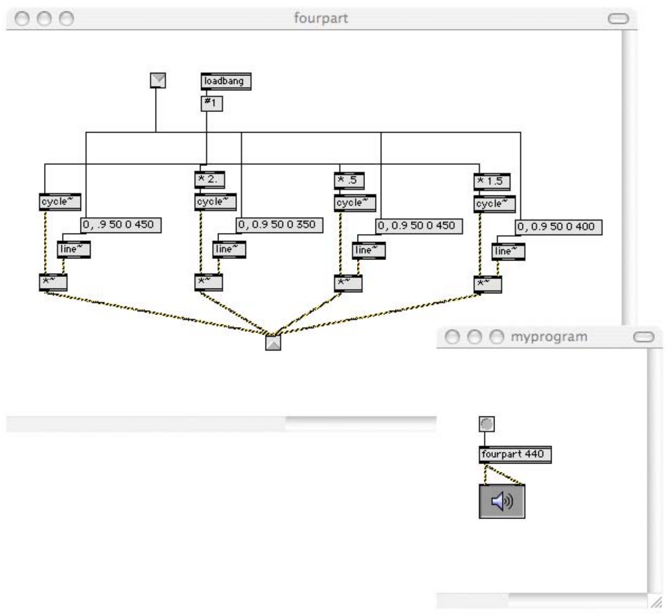

FIGURE 1
FOURPART HARMONY FUNCTION PROGRAMMED IN MAX/MSP

Musicians looking for programming tools in which to “code” sound synthesis are often drawn to a visual programming language, most likely either MAX/MSP or Pure Data (PD), an open source language idiomatically similar to MAX/MSP and maintained by one of MAX's creators, Miller Puckette [1][2]. The programming environment, when first started, looks very much like the blank screen of a text editor, a clean slate waiting to be filled. However, the program instructions are laid out on the screen in an even less linear way than most structured text based languages would be in a text editor window (Figure 1). Procedures, say for generating a sinusoid or the operation of addition, are drawn onto the screen inside of a box. The procedure names themselves can be chosen from a list of available primitives, somewhat freeing the programmer from needing to memorize the language's lexicon as well as facilitating exploration of available functions and their effects (“hmmm... I tried a *sin*, now what does this *cos* thing do?”) Other boxes may contain numeric constants, have special functions such as toggle switches or a “bang” that sends a signal to trigger an event or display graphical information such as waveforms (in an example of output) or envelopes (in an example of input). Such boxed procedures are often functions in the sense of taking one or more arguments and returning some output. Graphically, these input parameters and output return values come and go to other boxes, such as a box representing a digital-to-audio converter (ie. for play through the soundcard), the boxes connected with an adjoining line. For instance, addition takes two input lines and provides a single output line. The graphical metaphor of programming as plumbing system, a schematic of faucets, pipes and sinks, has long proven itself via MAX/MSP and PD. Programs created this way are used in music heard on the radio, movie theaters, clubs, concert halls and public art installations and have been extended to applications such as controlling theater lighting; MAX programs have been written that take data read off external sensors such as anemometers used in a public art display in Seattle Public Library’s Ballard Branch, processing the numbers as part of algorithmically driven musical composition. From the point of view of enabling musicians who may know no other programming language, PD and MAX/MSP are successful. Students without formal computer science training or knowledge in other programming languages regularly create nontrivial programs (known as patches in MAX/MSP and PD argot) that perform synthesis, time-frequency transformations, event handling and filtering. However this idiom is not the technology of choice for more general programming tasks: device drivers, web servers, 3D simulations of submarine telemetry are not written in this idiom. MAX/MSP cannot be written in MAX/MSP. Indeed MAX/MSP and PD offer hooks for extension via C/C++ for bolder explorations and customizations.

It is into this context that we introduce LogoRhythms, a music synthesis, computer audition API built on top of the functional flavored, typed-text paradigm of the UC Berkeley Logo interpreter. The UCB Logo interpreter, and therefore LogoRhythms, may be run either in command-line mode, via scripts or a combination of both where script code is read into the environment and available for use via the command-line. Either way, sound synthesis, for instance by combining and manipulating arrays of waveforms, or computer audition, such as descriptive statistics of a wave's spectrum, involves typing structured code at a prompt or into a text editor and subsequently running programs. LogoRhythms’s design is not meant simply to dumb down and simplify successful languages like Java or C++; although, arguably Logo is designed with a number of laxer syntactical conventions (ex. case-insensitivity) and more relaxed typing rules. Logo substitutes the morass of scoping rules typical of an object-oriented language-- appreciated by the more experienced programmer but sometimes mystifying to the neophyte-- for simpler procedure level rules. Similarly, Logo avoids explicit use of pointers and/or references, often sources of confusion for even experienced programmers. While Logo does accomplish an easing in rules along each of these lines, LogoRhythms should not be regarded as mere dumb-down. As a child of Lisp, Logo, and hence LogoRhythms, emphasizes programming in a functional style, a style weakly stressed at best in languages like Java, C or C++, although not useless there. Preparing students for functional programming fits well with other research projects in the University of Victoria’s joint CSc-Music program where some efforts examine the potential of contemporary functional languages such as SML in audio applications.

Just as the selection of Logo on which to the build LogoRhythms API reflects a desire to move neophyte programmers towards use of lower level “high-level" languages, the content of the API pays less focus to standards such as MIDI or the manipulation of prerecorded samples and more on manipulating lists and arrays of audio data represented as floats (though the LogoRhythms's programmer does not need to explicitly differentiate between floats and other number types). The emphasis on manipulation of arrays of numeric data differentiates LogoRhythms from APIs such as the javax.sound package, a colletion of functionalites that includes a prepackaged synthesizer and procedures for handling MIDI. In an acoustical analogy, javax.sound is like being handed a guitar while LogoRhythms is like being handed some wood and nylon polymer. Out effort is to facilitate first try manipulations of these raw materials where the student might be building a guitar, a lute, a ukelele or some other yet thought of instrument as well as the song played on the instrument.

ARCHAEOLOGY OF A COMPUTER LANGUAGE: WHY LOGO?

When considering where to build the LogoRhythms API, several languages were examined as a possible basis: Java, SML and Smalltalk for instance. Implementations of Smalltalk, particularly Squeak, already have rich high level audio support such as synthesizers or MIDI, a long history of use in educational computing and an easily extensible virtual machine environment. However, Logo was finally chosen for several reasons. First, the availability of a stable open source Logo interpreter in the form of the UCB Logo version 5.5. Second, the reasons already given as a language that encourages functional programming. Finally, Logo occupies an interesting historical niche in educational computing, sometimes even polemic. An archaeology of the language, ie. reexamining the design decisions behind its syntax and environment by actually using the language, has acted as a fulcrum for an interesting question in human-computer interaction. The language was originally designed at MIT's AI Lab as a tool to teach programming to perfectly typical primary school students, an intention which it successfully fulfilled [3]. Since Logo's inception in the late sixties, HCI (Human-Computer Interaction) has seen a considerable transformation with the notion of widespread computer literacy being supplanted by efforts in the possibly dumbing down concept of "user-friendly" and highly constrained graphical user interfaces. Although not further addressed in this paper, the archaeology alluded to here asks the question why typed-text, structured programming that was once taught to fifth graders (including these authors) is now considered almost solely the domain of first year university students in computer science and engineering?

Logo has long been used as a teaching language, though its applications are not limited to pedagogy. Logo based lessons have at times included musical and audio examples [4]. Early Logo teachers Michael Tempel and Mark Guzdial both have written of their efforts to teach programming through music using Logo, Guzdial providing anecdotes of his experiences with second, third and fourth graders who, he claims, were often more likely to debug audio errors than graphical errors as well as more likely to use subroutines to organize their programs [5][6]. Their examples, however, build up from a simple *tone* procedure as the sole sound producing primitive. Peter Desain and Henkjan Honing provided a very eloquent scoring language in Logo called LOCO [7]. Their language's aim was to "enable a composer to express ideas in a direct way." Their language therefore is rich in the argot of traditional musical theory, composition and structure. They justify their choice for Logo saying, "This language [Logo] ... had the enormous advantage of being easy to learn." They note of their experiences with LOCO, "We have used LOCO in a number of workshops. It has proven to be a rich, motivational context for different kinds of participants. After a short explanation they were able to start expressing their own ideas, depending on previous knowledge and experience in the field of traditional or computer music." Their language was also used in their university level courses on computer music. LogoRhythms has the advantage over older versions of Logo at coming at a time when sound on the personal computer has greatly improved-- owing to better sound cards, algorithms and processing speed. Therefore, it can accomplish far more complex timbres and rhythms than Tempel or Guzdial had hardware to produce. Furthermore, much more than LOCO, LogoRhythms's emphasis is on a lexicon taken from signal processing, not music theory. The student may create the latter connections themselves within their programs. Early versions of UCB Logo had essentially no audio support. Version 5.5 contains nothing more than a simple *tone* procedure that LogoRhythms replaces with its own, more configurable, *TONE*. We hope LogoRhythms provides a simple entrance into programming computer music and programming in general while focusing on low level audio manipulations.

A BRIEF TOUR OF THE LOGORHYTHM'S API

Logo's accessibility for the new user, even very young ones, is enabled in a variety of ways: garbage collection, dynamic binding, case insensitivity, and, very importantly, allowing more than one way to do something. LogoRhythms's design attempts to maintain this spirit. Its procedures often overlap in their functionality with simpler and more complex versions coexisting. Thus an easy foothold is provided for the first time user-- but with room to grow.

Sound producing procedures are divided between two groups: those that operate on wavetables and send their output to the soundcard and those that allow the programmer to manipulate arrays of sound data, playable via a *PLAYWAVE* procedure. (Please note that italics indicate a Logo or LogoRhythms procedure name. Logo itself is case insensitive so *harmony* and *HARMONY* and *HaRMony* all point to the same procedure. The parameters have been omitted here but are described in the LogoRhythms's documentation.)

Wavetable procedures include *TONE*, *SOUND* and *HARMONY*, each a slightly more feature rich version of the former. Each procedure is ultimately playing sinusoids, the programmer setting some combination of frequency, duration and envelope. For instance:

sound 440 [[.9 50] [0 450]]

With this procedure call, a sinusoid of 440 Hz is played for 500 ms by linearly ramping to .9 times full volume in 50 ms and then decaying to silence in 450 ms.

The other suite of procedures are true functions, generally returning arrays. The programmer starts by creating a single period of some waveform: *SINEWAVE*, *TRIANGLEWAVE* or *SQUAREWAVE* and then manipulates these building blocks with functions such as *COPYWAVE*, *WAVEENVELOPE*, *COMBINEWAVES* and *CUTWAVE* among others, each returning an array.

The two sets of procedures are fused with calls to *TONEWT*, *SOUNDWT* and *HARMONYWT* which operate similar to their simpler, non-*WT* forms but allow the user to specify their own array to be used as the wavetable.

Thus far the procedures discussed control frequency related features of sound: timbre, pitch and harmony. What about the "rhythm" component of LogoRhythms? Rhythm, the pattern of pulses or notes within the composition, is controlled by the structure of procedure calls, for instance embedding them in a *REPEAT* loop or by the order of function calls. The duration of a sound is specified by its amplitude envelope. In other words, the program structure provides the rhythm's structure. LogoRhythms is without a scheduler and while all time parameters are specified in milliseconds, actual performance will vary from machine to machine. However, LogoRhythms does include several procedures useful in creating rhythmic patterns. Perhaps most important is *REST,* a procedure that essentially produces a note of zero amplitude for a specified number of milliseconds. To aid in using arguments with different time signatures, *SETTIME* can be used to normalize the arguments to a standard time.

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

Here the envelope would have been played for 1000 milliseconds (50 + 950). *SETTIME* normalizes the total length of *new_envelop* to 500 milliseconds.

Turtle graphics has long been a core feature of the Logo language. Indeed one might view Logo as a drawing language where program instructions command a cursor known as the turtle to sketch. Logo creators like Seymour Papert saw geometry and drawing as an excellent first application area to start programmers, an area rich in mathematical opportunity but where debugging could be aided by the students' already well developed sense of space and how their own bodies fit into that space.[3] An error in a rendered drawing might even be paced out across the floor, the code becoming instructions for an interpretive dance. While LogoRhythms shares the view of programming as a creative act, perhaps it's harder to make the same parallel arguments in an audio analogy. LogoRhythms does use turtle graphics by way of a Logo library procedure, *DRAWWAVE*, that allows one to view audio signals, spectrums or really any array of numeric data thus providing basic oscilloscope functionality. Since UCB Logo is designed to run on a variety of platforms including OSX, Windows and Linux/Unix, the compromises between graphic environments keeps the output basic. As for interpretive dance? The analogy may hold less well than the simple architectural ones of square houses with isosceles triangle roofs given by Papert, but it's not unreasonable to make a connection between the smooth changes of a sinusoid and an ocean swell inspired hula, the discontinuities of a triangle wave and the see-saw of a tango or a noisy, spiky square wave and the jumping, energetic hip-hop of highly percussive dance music.

Most of these procedures are implemented as primitives in C directly as part of the UCB Logo interpreter. Others are implemented in the Logo language itself and provided as Logo library procedures. For instance, *FFT* is implemented in C within the interpreter; *SPECTRUM* is implemented in Logo as a library procedure. All of the code is available as open source and therefore, inspectable by the student, useable as a model for the student's own programming. Indeed, the availability of the open source interpreter from the UCB maintainers eased this project considerably. Logo, both historically and through modern commercial implementations such as LCSI's Micro Worlds, usually is associated with constructivist education. Providing modifiable examples via open source code balances this formula, allowing deconstruction, dissection and contextualization as part of the modeling process of constructing new knowledge and skills.

The full LogoRhythms source code and API documentation can be found at www.sanitysewer.com/LogoRhythms (as of spring 2006).

PROGRAMMING IN A FUNCTIONAL PARADIGM

Functional programming refers to a structured program where the primary structural unit is the function. The function in a functional programming idiom resembles that used in mathematics: they always take an argument and they always return a value, for example, the identity function $f(x) = x$. Global variables are rare or completely absent in true functional languages; procedures do not modify variables outside their local scope nor return void. The lack of globals is a major appeal of functional programming, helping eliminate untoward side-effects as potential bugs. Loop statements are often absent or downplayed in favor of recursion. While not requisite, many functional languages use lists, a data structure that lends itself to recursion. While functional programming can be accomplished to some degree in the most widely used languages such as C or Java, famous functional languages include Lisp and SML. Nyquist is a flavor of Lisp specifically created for audio applications that demonstrates one style of functional programming in musical composition [9]. Logo is not a strict functional language, although an offspring of Lisp. However, Logo encourages functional programming and relies heavily on lists. LogoRhythms similarly stresses functional programming.

While removing globals does remove one possible source of errors, the data must still be available to a procedure, fed to the function via its arguments. In the absence of structures like structs or classes, these arguments are likely to be lists, lists of lists, lists of lists of mixed primitive types, etc. The tidiness of the functional paradigm can quickly start to suffocate under long gangly arguments. The remainder of this paper provides further introduction to LogoRhythms by more closely examining how higher-order functions, lists and encapsulation can tame otherwise unwieldy arguments. It is also here by introducing students to concepts such as lists, encapsulation and recursion, that LogoRhythms begins to make the connection for the student between simply composing sounds and music and the larger world of computer science and programming.

PRETTYING UP THE ARGUMENTS

Let's start with an example of what might be considered an ugly argument using LogoRhythms's harmony procedure.

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

This procedure will play a note for half a second comprised of sinusoids tuned to four separate frequencies. The fundamental frequency could be considered 440 Hz. The envelope that follows "440" instructs *HARMONY* to linearly ramp up to .9 times full volume in 50 ms and then linearly decay to zero volume in 450 ms. This procedure can be made cleaner, ie. reduce the need to directly handle the morass of nested lists, and more useful by employing higher-order functions, lists and encapsulation.

Encapsulation, in a broad sense, refers to the containing, even hiding, of information through scoping rules. Take for example the list that makes up the argument to *HARMONY*. This list can simply be encapsulated inside of another function. The list data is local to the second function and returned by it. Of course, it's not hard to extend the functionality of this second function such as adding parameters that modify the list to be returned. Here's a function called *fourpart* that will produce a list parameter for the *HARMONY* primitive. This is the same procedure shown in the MAX/MSP abstraction of figure 1.

```
to fourpart :freq
  local [a]
  make "a []
  make "a fput list freq     [[.9 50] [0 450]] a
  make "a fput list freq*2   [[.9 50] [0 350]] a
  make "a fput list freq*0.5 [[.9 50] [0 450]] a
  make "a fput list freq*3/2 [[.9 50] [0 400]] a
  output a
end
```

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

```
harmony fourpart 440
```

Ostensibly, this is a much clearer semantics. If the student programmer-musician also implement *fourpart*, even better. This first example demonstrates function composition of the form *f(g(x))* where *f(x)=HARMONY g(x)*, *g(x)=FOURPART x* and *x=440*. This same example is shown in figure 1 as programmed in MAX/MSP.

Templates are UCB Logo's device to allow the use of anonymous functions or, more specifically, lists of instructions [4][8]. The real flexibility of templates begins to be realized when examining UCB Logo's *APPLY* function. *APPLY* itself takes a function as its first argument. The symbol "*?*" is called an explicit-slot and marks the parameters of the template function. The code:

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

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

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

This recursive procedure is very similar to map functions found in many functional languages. Its first parameter is a template. The second parameter is a list of arguments to the anonymous function (ie. the template). It will recursively traverse over the list "a.list" applying each value in the list to the anonymous function "a.func." It differs from other map functions in that nothing, such as a new list, is returned since it's intended to be used with an IO affecting anonymous function. Our last code example uses *to sing* with the previous *harmony fourpart* demonstration to create a simple composition.

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

Do re mi fa so la ti do.

Using templates in this manner is similar to the use of lambda expressions in Lisp and the semantical distilling demonstrated here with LogoRhythms can be analogously accomplished using lambda expressions in a Lisp based language such as Nyquist [9].

CONCLUDING COMMENTS

The world of computer music is not short on excellent software for audio synthesis and analysis, including programming languages. The niche that LogoRhythms seeks to fill is that of a typed-text, structured programming language that allows the student musician-programmer to explore low level audio synthesis in a powerful, but forgiving language, a language that is to be viewed as a stepping stone into flexible and widely used high level languages such as Java or C++ or even audio specific languages such as Chuck [10]. Today, many general programming problems that comprise many classroom programming exercises appear solved, but the computer still exists as a useful modeling tool, and modeling requires programming of some sort. Musical composition is a creative endeavor and as such in this context, an essentially endless possibility of unique programs. LogoRhythms is an API designed to help bridge the neophyte programmer into the wider world of computer programming, both in audio and other applications.

While building this API, we had two audiences in mind. The first is the group of musicians looking to crossover and improve their engineering skills while participating in the university level computer-music interdisciplinary curriculum. The second is the audience who historically motivated the creation of the Logo programming language: primary and secondary school students. This latter group brings up interesting human-computer interaction and pedagogical questions of why computer literacy skills that were making inroads in primary school education a quarter century ago are now seen as material for first year university level students majoring in computer science? Our first stab hypothesis doubts that computers proved too hard for younger students. With the LogoRhythms API stable, we now hope to move it into the field, trying it as the tool it was designed to be as well as gathering user feedback in addressing more theoretical questions of computer literacy. Inquiries from educators wishing to collaborate or just try LogoRhythms are encouraged.

REFERENCES

[1] Puckette, M, "Max at Seventeen", *Computer Music Journal*, Vol 26, No 4, Winter 2002, pp 31-43.

[2] Zicarelli, D, "How I Learned to Love a Program that Does Nothing", *Computer Music Journal,* Vol 26, No 4, Winter 2002, pp 41-51.

[3] Papert, S, *Mindstorms: Children, Computers and Powerful Ideas,* 1980.

[4] Harvey, B, *Computer Science Logo Style, Volume 3: Advanced Topics,* 1987.

[5] Tempel, M, "Logo Music Tools", *Logo Foundation Online Papers: http://el.media.mit.edu/logo foundation/pubs/papers,* accessed March, 2006.

[6] Guzdial, M, "Teaching Programming with Music: An Approch to Teaching Young Students About Logo", *Logo Foundation Online Papers: http://el.media.mit.edu/logo foundation/pubs/papers,* accessed March, 2006.

[7] Desain, P and Honing, H, "LOCO: A Composition Microworld in Logo", *Computer Music Journal,* Vol 12, No 3, Fall 1988, pp 30-42.

[8] University of California, *Berkeley Logo Usermanual,* 1993.

[9] Dannenberg, R B, "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis", *Computer Music Journal,* Vol 21, No 3, Fall 1997, pp 50-60.

[10] Wang, G and Cook, P R, "ChucK: A Concurrent, On-the-fly, Audio Programming Language", *Proc 2003 ICMC*, 2003, pp 1-8.