Program Auralization: Sound Enhancements to the Programming Environment

Christopher J. DiGiano
digi@dgp.toronto.edu

Ronald M. Baecker
rmb@dgp.toronto.edu

Dynamic Graphics Project
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

Sound has the potential to improve our understanding of a program's function, structure, and behavior. In this paper we identify classes of program information suitable for mapping to sound and suggest how to add auralization capabilities to programming environments. We describe LogoMedia, a sound-enhanced programming system which illustrates these concepts.

Keywords: Program Auralization, Non-Speech Audio, Software Visualization, Programming Environments

Program auralization

Gerald Weinberg, author of The Psychology of Computer Programming, describes programming as "at best a communication between two alien species" (Weinberg, 1971). Indeed, despite efforts in the field of software visualization, programs are in desperate need of better means of presentation and clarification. This paper explores the potential of non-speech audio to increase the communications bandwidth between the two "species."

Non-speech audio is quickly becoming an integral part of the computer's ability to record and present data, as is evidenced by recent introductions of computer systems with built-in high quality sound input and output from graphics workstation manufacturers such as Sun Microsystems (Yager, 1991) and Silicon Graphics (Smith, 1991). Interface designers have only just begun to tap into the capabilities of this new hardware. Although sound has seen limited use in some software visualization systems (Baecker, 1981; Brown, 1988), it is only recently that computer audio has been seriously applied to the programming domain to help elucidate the behavior of running programs (Jackson and Francioni, 1992; Sonnenwald, et al., 1990). *Program auralization* refers to the use of non-speech audio for supporting the understanding and effective use of computer programs.

The first part of this paper discusses some properties of sound that are useful to the human-computer interface. We then propose a program auralization taxonomy describing how sound may be associated with execution behavior and with the structure of the code. For each of the three categories in the taxonomy we discuss the relevant characteristics of sound and useful types of program auralization tools. We conclude by describing our implementation of a sound-enhanced Logo programming environment called LogoMedia.

Non-speech audio at the interface

Sound has unique properties which can be exploited in the computer-human interface. For certain types of data sound is the most intuitive means of understanding the information. Experiments mapping time-varying economic data to sound have found that humans can more effectively identify correlations using sound than with graphics (Mezrich, Frysinger, and Slivjanovski, 1984). Logarithmic data, normally difficult to perceive graphically, has also been shown to benefit from audio portrayals since pitch and loudness are logarithmic functions of frequency and intensity (Buxton, Gaver, and Bly, in preparation, ch. 3). Gaver has demonstrated the ability of "everyday" sounds to alert users instantly yet unobtrusively about certain events (Gaver, 1989; Gaver, 1990; Gaver, 1991a; Gaver, 1991b).

Sound can be used to relieve the burden of the visual interface. With the increasing popularity of graphical interfaces, more and more applications are using images to portray information. The resulting collage of windows and colors has the potential of overloading the human visual system. The computer industry is also moving toward smaller, more portable computers with displays limited by current technology to fewer colors, less pixels, and slower update rates. The effective use of sound offers an attractive design solution to problems introduced by both trends, providing a new output modality to complement the graphical interface.

The multi-dimensionality of sound makes it useful for presenting complex data which is otherwise difficult to represent. Sound can be systematically varied across many dimensions including loudness, pitch, vibrato, rate of modulation, timbre, and tempo (Buxton, Gaver, and Bly, in

preparation). Some works (Yeung, 1980) suggest human can differentiate sounds of up to 20 dimensions. Another useful property of sound is that humans do not necessarily have to be facing the source in order to hear it. This means that the focus of attention can be applied to the computer screen, a printout, or even a cup of coffee while continuing to process auditory information.

A program auralization taxonomy

Auralizations appropriate to the programming domain can be divided into three major categories according to the characteristics of the information streams mapped to sound sequences. Each sound generated as part of an auralization corresponds to some item in a time varying stream of data. We characterize these streams by their *sound generators*—the activity which determines which datum comes next for the purpose of auditory mapping. Probably the most obvious information stream is the sequence of program states during execution. In this case the sound generator is the running program itself. As shown in Table 1 we have identified two other unique sources of program sounds. Each activity corresponds with a program development phase and covers a particular type of program information.

Phase	Sound Generator	Program Information
execution	executing	variables, internal state, control flow, backtracking
review	scrolling	modules, goals/plans, keywords
preparation	parsing	syntactic structure

Table 1: *Three categories of program information which are candidates for auralization*

Execution

The execution of a program causes variables and machine state to change over time. By mapping this data stream to sound the programmer can listen to his or her code run which has the potential to reveal useful information about the behavior of the program.

Review

A programmer examines his or her code by scrolling through the text in a window. The sequence of program sections encountered can be mapped to sound, helping provide contextual information while using the visual information from the screen for detailed assessment.

Preparation

While compiling or entering a program, the computer evaluates language tokens and identifiers in a specific sequence not necessarily corresponding to the stream of characters making up the code. This sequence can be translated into sound, allowing the programmer to monitor the progress of such activity.

An Example

Consider the following example programming session in which we auralize the process of preparing, executing, and reviewing a procedure to compute the factorial function. Figure 1 shows one way of performing this calculation using Logo (Harvey, 1986). A single call to factorial generates a series of nested recursive calls which multiplies a series of numbers decreasing by one from the original value to one.

```
1  to factorial :num
2    if :num = 0
3      [output 1]
4      [output (:num * factorial (:num-1))]
5  end
```

Figure 1: *Logo code for computing factorial*

While typing line 4 we must be careful to match all of the nested delimiters. Using a standard editor for entering the code, we could easily skip one of the right parentheses. A sound-enhanced programming environment in which sounds are generated during program entering can inform us immediately of such an error. Typing the left bracket might initiate an unobtrusive continuous background sound such as the soft ticking of a clock as depicted in Figure 2. The following left parenthesis might start the background sound of a bubbling fish tank which is layered on top of the ticking noise. A fan noise might result from typing yet another left parenthesis. Each background sound continues playing until its respective matching delimiter is typed, so that by the end of the line none of the three channels should be audible. Now if we skipped one of the right parentheses we would reach the end of the line without the bubbling stopping—an obvious indication of a problem.

Suppose we mistakenly implemented factorial as shown in Figure 3 without the base case, resulting in a program which runs forever or at least until stack memory is exhausted. When we call our flawed factorial procedure from the Logo interpreter with the argument of 20, the computer would pause for a while and finally produce an error message. Suppose we suspected the problem was actu-

```
[                                        ticking
[output (                                ticking and bubbling
[output (:num * factorial (              ticking, bubbling and fan
[output (:num * factorial (:num-1)       ticking and bubbling
[output (:num * factorial (:num-1))      ticking
[output (:num * factorial (:num-1))]     silence
```

Figure 2: *Delimiter matching sounds while entering line 4 of the procedure in Figure 1*

ally with the value of the parameter `:num`. Was it actually decrementing for each successive call, or was the computer using the outermost runtime stack frame to determine the value of `:num`? Using a sound-enhanced programming environment we can monitor the execution behavior of this and other parameters. We might associate a continuous sound which varies in pitch with the value of `:num` as the program executes—the lower the number, the lower the pitch. Now when we execute the program, we would hear a note which starts high but quickly drops in pitch. It would be apparent that `:num` is indeed being decremented and our error must be elsewhere.

```
1 to factorial :num
2     output (:num * factorial (:num-1))
3 end
```

Figure 3: *Factorial procedure with missing base case*

If the factorial procedure were part of a larger program it might help us to understand the code if we knew how often factorial gets called and by which components. A sound-enhanced programming environment can be asked to generate a sound each time the identifier `factorial` scrolls through the editor window. The sound warns us in advance to pay closer visual attention to the code coming into view.

Related audio applications

In recent years the use of non-speech audio at the interface has become a serious study for interface designers. A seminal work by Gaver (1989) was the Sonic Finder which used everyday sounds to indicate common operations in the Macintosh direct manipulation interface. Dragging the mouse generated a scratching sound. The act of deleting files by dropping their desktop icons into the trash emitted a crash. The mouse cursor at times also acted like a mallet, creating sounds as the user clicked on various icons on the screen. Large file icons had a low hollow sound, while smaller files had a higher pitch.

Following the Sonic Finder were ARKola (Gaver, 1991a) and EAR (Gaver, 1991b) which used sophisticated event-driven sounds. ARKola simulated the sounds of a collection of bottling plant machines running simultaneously in various stages of disrepair. EAR sounds included paper falling, the shuffling of people gathering in a room, and the pouring of a pint of beer to remind users of various events and conditions taking place or about to take place at Xerox EuroPARC.

Other studies have focused on using sound to represent complex computer data, referred to as *data auralization*. Exvis (Smith, Bergeron, and Grinstein, 1990) used both graphics and sound to portray various types of data including magnetic resonance scans which varied across many dimensions. As the user moved the cursor over the graphical representation, data points emitted a characteristic sound with frequency and intensity related to particular dimensions of the sample. Bly (1982) tested human analysts on six-dimensional data which she mapped to six sound characteristics: pitch, volume, note duration, fundamental waveshape, attack envelope, and overtone waveshape. She found that the information content of the data could indeed be enriched through audio.

As an aural alternative to graphical icons Blattner, Sumikawa, and Greenberg (1989) proposed "earcons" for denoting a variety of computer events. Earcons consisted of single notes, short melodies or combinations of other earcons. The authors suggested that properly designed earcons could be learned quickly and associated with arbitrary objects and computer operations.

Recent works have demonstrated the potential of audio in the programming domain. Jackson and Francioni (1992) used audio to improve the programmer's awareness of the behavior of parallel programs by generating sounds based on trace data recorded during execution. Program events being monitored caused a unique note or melody to be played using a particular timbre mapped to each of 16 processors. Music theory was used to identify the most appropriate melodies for each event. Sonnenwald, et al. (1990) developed a set of primitive function calls for incorporating sound in program code for the purpose of elucidation. Although the authors primarily discussed the capabilities of their system for portraying parallel programs, its general audio functions could be applied to a variety of programs both concurrent and non-concurrent. While animating algorithms Brown and Hershberger (1991) generated sounds corresponding in pitch to elements being inserted into a hash table, items being sorted, and to the number of active threads. The sound of a car crash we used to indicate hash collision. Brown and Hershberger identify four main uses of sound in algorithm animation: "reinforcing visual views, conveying patterns, replacing visual views, and signaling exceptional conditions."

Sound in the programming domain

Most of the above examples deal with sound in specific applications. In contrast this paper discusses the uses of non-speech audio in a much more generic context—the programming environment. Auralization in the programming environment is a challenging interface design problem since the sounds must be adaptable to a variety of conditions. A sound-enhanced programming environment must represent both data and events aurally. A further complication is that the exact types of data and events are unknown, since the programming environment is meant to be used to develop arbitrary applications using a variety of data and control structures. Finally, depending on the stage in the program development process, the programmer may use sounds in different ways. When debugging, for instance, the programmer may want to introduce generic or symbolic sounds quickly to determine the behavior of his or her program. When creating auralizations for presentation purposes, he or she may want more specific or iconic

sounds to portray more appropriately the program execution.

Figure 4 compares non-speech audio applications based on the type of information they represent using sound as well as their *articulatory directness.* Articulatory directness as defined by Hutchins, Hollan, and Norman (1986) is the degree to which form follows function in an interface. Gaver (1989) used this measure to describe the intuitiveness of the perceptual mappings used to link the "model world" of the computer and the audio display, with *symbolic* being the least intuitive and *iconic* the most. A symbolic mapping is an arbitrary association made between sound and computer information, whereas an iconic mapping is based on well-known physical properties. Exvis and Bly's work which focus on data auralization belongs near the symbolic end of the articulatory directness scale because they relate properties of sound such as pitch and loudness to data which has no inherent connection with sound. ARKola and EAR on the other hand use iconic mappings because of the obvious associations between events and their sounds.

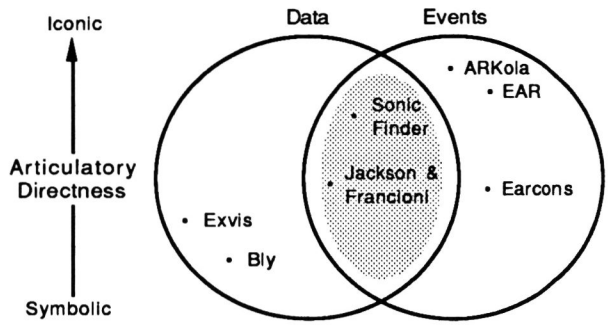

Figure 4: *Comparison of non-speech audio applications*

The intersection of the two circles in the figure contains those applications using non-speech audio to represent both data and events. The Sonic Finder belongs in this category because of the way impact sounds suggest the type and size of files. Another application for this category is program auralization systems which portray both data and control flow using sounds. A sound-enhanced programming environment, represented by the shaded area of the figure, must use sound to portray both data and events. An area is more appropriate than a point because the programming environment must be capable of using sound to portray a variety of different types of data and events at different levels of articulatory directness.

Execution sounds

Comprehending the course of execution of a program and how its data changes is essential to understanding why a program does or does not work. Auralization expands the possible approaches to elucidating program behavior.

Protocol studies of novice, intermediate and expert programmers debugging code suggest they make use of a variety of techniques for observing the changing values of variables and monitoring sub-program calls (Nanja and Cook, 1987). As shown in Table 2 we can divide execution information into values and events. Value information refers to the contents of data structures as they change during execution, often known as data flow. Event information is the stream of operations on these data structures as well as the flow of control from one line of code to the next and from one sub-program to the next.

Information Type	Values	Events
common	queue size, tree depth	loop, branch, push, pop, search, sort
arbitrary	a, b, c	sub-program calls, sub-program returns
internal	call stack size, memory space	register usage, backtracking

Table 2: Candidates for auralization during execution

For both values and events we group the execution information into three categories: common, arbitrary, and internal. *Common* information refers to typical data and control structures which we can anticipate will be the subject of interest to the programmer. Sound enhancements to the programming environment should include default methods for monitoring the status and usage of these structures. *Arbitrary* execution information refers to the types of data and control flow which cannot be predicted. For this type of information a sound-enhanced programming environment should provide tools for establishing custom auralizations. *Internal* information is the changing machine state over the course of execution which is largely programming language and machine dependent. Default auralization for this type of data can increase the programmer's awareness of activities such as resource usage occurring behind the scenes.

Monitoring execution values

Traditional techniques for monitoring values during execution generate text to indicate the current program state. An obvious method of displaying text relating to program variables is to insert output statements directly into the source code. Debuggers allow the values of variables to be printed without having to modify the source code. A popular debugger for the Unix environment, `dbx` provides a set of commands for maintaining a list of expressions containing program variables to be printed during execution.

Software visualization systems use pictures to present often more complex program data in ways which are simply not possible using text. Visualizations are useful for making abstract program information easier to understand. The University of Washington Program Illustrator, for instance, recognizes the abstract data structure for digraphs within Pascal programs and represents them on the screen as a collection of circles connected by arrows (Henry, Whaley, and Forstall, 1990). Pictures of programs are well-suited for displaying large quantities of computer

information simultaneously. In the film *Sorting Out Sorting* Baecker (1981) uses graphs to show the progress of several sorting algorithms on thousands of data items.

Data auralization provides a useful extension to textual and graphical techniques for monitoring execution. Although not suitable for conveying exact values, auralization can indicate trends and increase the number of dimensions capable of being presented simultaneously. Auralizations of program data generate sounds varying across dimensions of pitch, volume, reverberation, panning, envelope, and tempo in response to the changing values of program variables. An important advantage over visual feedback is that auralization frees the programmer to inspect the actual code while it is executing.

Sound can be useful in conveying simple auxiliary information relating to common data structures such as the size of a queue or the depth of a stack. Ideally, these auralizations of high level data structure values could be built into the language or class library. Thus, the user is free from having to specify the particular element of the data structure to map to sound.

For listening to arbitrary data a sound enhanced programming environment must support the ability to map changes in variable values to a variety of audio dimensions. The mapping must be flexible so that the user can experiment with various auralizations in order to find the one most appropriate for his or her application. Users should be able to select from a variety of synthesized and prerecorded sounds, or make their own sounds. To support this type of opportunistic auralization in LogoMedia we have developed an audio expressions tool for making data-sound associations.

Internal values in an executing program refer to machine states not normally known to the programmer. The size of the call stack and the amount of free memory are two examples of internal values. The default sounds available for internal values should easily fade into the auditory background of the listener. Patterson's study of alarms (1989) provides useful guidelines for designing non-obtrusive sounds such as the use of slow onsets and rhythmic patterns.

Monitoring execution events

Traditional techniques for monitoring execution events produce a sequence of text lines indicating control flow. The simplest method requires no additional programming environment tools: inserting lines of code before or after suspect statements which cause a message to appear on the screen. More sophisticated program tracing approaches are found in debuggers such as `dbx` which can automatically generate generic tracing information while simulating the execution of a program.

Software visualization systems use pictures or diagrams to indicate the execution path. Some visualization systems such the Transparent Prolog Machine (TPM) (Eisenstadt and Brayshaw, 1988) can graphically trace programs automatically. Other software visualization systems offer more customizable and therefore potentially more salient trace feedback than traditional debuggers. LogoMotion (Baecker and Buchanan, 1990) is a programming environment in which procedure calls can trigger tailored visualization code.

Computer audio can enhance textual and graphical execution event information by generating sounds in conjunction with the execution of a line or a collection of lines and by using multiple voices to indicate layers in a calling chain. As with program data, control flow auralization allows the programmer to focus his or her visual attention on the actual code being run. Tracing programs using audio can also uncover repeated patterns of program behavior. Jackson and Jackson (1992) noted that users of their system for auralizing parallel program events recognized "melodies" characterizing various communication patterns between processors. They also pointed out that missing or delayed parts of the pattern were noticeable.

A sound enhanced programming environment should allow users to associate sounds with common, arbitrary, and internal events. Common events include the execution of control structures such as loops and branches as well as operations on common data structures such as stacks and linked lists. For LogoMedia we have developed sounds of plates stacking, unstacking, and breaking for the typical push, pop, and overflow events. Additionally, there are sounds for indicating list operations using the metaphor of a three-ringed binder—popping open for an insertion, tearing for a deletion, and page flipping sounds for a search.

Arbitrary events of interest to the programmer might be calls to sub-programs, returns from sub-programs, or the execution of particular lines. Sound enhancements to the programming environment should include the ability to generate sound in response to the program counter reaching arbitrary lines of code. In LogoMedia entering or exiting a procedure can trigger auxiliary Logo code for turning sounds on or off or changing their quality. We are currently developing a tool for specifying program *audiopoints* instead of breakpoints.

The ability to monitor language or machine dependent internal events using sound is certainly worthy of further study. Low level internal information such as operations on registers and high level information such as Prolog backtracking may prove to be good candidates for auralization, since the programmer is more likely interested in the general pattern of activity, rather than exact events.

Review sounds

The review phase when the programmer interactively explores the source code is another opportunity for integrating sound into the programming environment.

Studies in program comprehension (Gellenbeck and Cook, 1991; Kesler and Uram, 1984) have reported the utility of information supplementary to the raw code in providing clues to help programmers understand a program and predict its behavior. Examples of this kind of information include indentation, comments, typographic signaling, mnemonic names, module organization, goals and plans, profiling

data. Sound offers a new modality for communicating this ancillary information. Unique to this phase is the method by which sequences of sounds are derived from the series of points in the code on which the programmer focuses his or her attention.

Visualizations of software can enhance the programmer's understanding of programs by providing a variety of views which either compress or elide uninteresting information. Baecker and Marcus illustrated a collection of information-rich typographic overviews in SEE (Baecker and Marcus, 1990). Small's static typographic visualization system, Viper, revealed the advantages of interactive systems for culling, clarifying, and amplifying parts of the program text interesting to the user (Small, 1989). Viper allowed the programmer to specify code filters used for remove certain lines of code and highlight statements in others.

Listening to source files

Computer audio offers an attractive alternative to textual enhancements to the code for conveying information about the structure or intent of a program. Sound-enhanced programming environments should allow portions of code at the focus of the user's attention to be mapped to sound. Perusing the code causes the system to generate a sequence of sounds, providing an auditory context for more detailed visual examination.

Auralizing the source code requires that interesting portions of the text somehow be identified for mapping to sound. This can be done automatically by parsing the code and flagging syntactic structures such as blocks, and procedures as is done with Viper. Profiling and verification systems such as Unix `prof` or `lint` can automatically associate other kinds of useful data with sections of code. A semi-automated alternative is using a structured editor which can identify syntactic structures and problem solving strategies. Finally, the programmer can manually select the sections of code for auralization. Although this last option may seem tedious, the programmer may be able to take advantage of auralizations already in place for the purpose of monitoring events during execution. If the programmer had been thorough in marking a variety of interesting program events with audiopoints, the events themselves can serve as "audio landmarks" (Jenkins, 1985) into the code—a form of audio documentation.

In LogoMedia we are developing the ability to generate audio feedback while scrolling through a Logo document. As interesting items enter and leave the window view the sound will change accordingly. Program constructs identified during interpretation such as procedure definitions, Logo lists, and lines with audiopoints will each have characteristic sounds. Programmers can perform an auditory search of their code by asking the system to generate sounds if the items in focus meet certain criteria such as if they are an identifier of a certain name or a particular Logo command.

Rapid scrolling through a program may generate emergent sounds patterns indicative of particular types of code or a certain style of programming. Speeth (1961) found that subjects were able to discriminate between earthquakes and explosions using compressed auralizations of large quantities of seismic data which were not visually distinct. Because of sound's ability to reveal patterns in massive quantities of data, new insights may result from the rapid audio review of complex program.

Preparation sounds

The process of preparing code for execution can be mysterious to the programmer. In a sound-enhanced programming environment the order in which programming language constructs are parsed can be a source of sound sequences and increase the programmer's awareness of this ongoing activity. While entering a program syntax-directed audio feedback can help identify syntactic errors. During compilation sounds can be used to monitor progress.

Syntax-directed editors attempt to reduce obvious syntactic errors through typographic feedback during code entry. In some systems keywords recognized by the editor change font after the programmer types a separator such as the space bar. Lines following an IF clause are automatically indented until the block is terminated with a semi-colon. An obvious problem with syntax-directed editors is that they force users to adopt a particular typographic style. The persistence of typographic syntax-directed feedback may also be an annoyance to programmers. The style of the program text remains changed even after the words have been typed and the programmer is sure it has been entered properly.

Listening to program entering

A sound-enhanced programming environment offers the programmer an alternative output modality for conveying similar syntactic feedback during program entering. An approach we are investigating for LogoMedia is to provide subtle background sounds to indicate whether the computer recognizes program constructs which are typed and to differentiate aurally between classes of syntactic structures. Word classes might include operators, built-in procedures, control constructs, previously declared procedure identifiers, previously declared variable identifiers, new procedure identifiers, and new variable identifiers. Discrete sounds are triggered by completing a typed word with a delimiting character such as the space bar or comma. After a fixed period of time the sound terminates. Following Patterson's approach to reducing obtrusiveness (Patterson, 1989), each sound conforms to an intensity envelope in which the sound begins quietly, is sustained for brief moment, then fades to silence. We are also devising continuous sounds to demark sections of code and help identify problems such as wrong number of delimiters or wrong number of parameters. When the programmer starts a Logo block with a left bracket, for instance, a sound is generated until the matching bracket is typed. Nested

sections of code cause collections of continuous sounds to be layered.

Listening to compilation

A programmer typically glances periodically at textual output from a compiler during the translation process. A sound enhanced programming environment should support the ability to generate sounds in response to a compiling program to prevent the user from having to shift visual focus from other activities. In some programming environments such as the Macintosh Programmers Workshop sounds are used to differentiate successful from unsuccessful compiles. However, more sophisticated uses of sound can undoubtedly convey even more than binary status information. In the "early days" of computing intrepid hardware hackers on the TX-2 attached a speaker to the index registers. The resulting sound with some training was used by programmers to determine the state of running processes such as compiles which had limited visual feedback. Richer sounds could be based on the same types of audio feedback used during the interactive parsing of programs as mentioned in the previous section. An interesting question is whether meaning could be extracted from the emergent sounds generated by this high speed parsing of program structure. Since Logo is an interpreted language, LogoMedia has no facilities for compilation-based sounds.

LogoMedia

We have implemented some of the auralization techniques outlined above relating to the execution and review of programs. Sound capabilities were added to a programming environment for software visualization developed here at the University of Toronto called LogoMotion (Baecker and Buchanan, 1990). The goal of the new version dubbed LogoMedia is to provide the programmer with simple, non-invasive[1] techniques for using sound to aid in the development and presentation of arbitrary programs. We are not trying to enforce a particular use of sound for auralizing programs, but rather provide a variety of tools for the programmer to use sound easily while developing.

LogoMedia does not actually synthesize its own sounds but rather sends Musical Instrument Digital Interface (MIDI) (IMA, 1983) messages to the Macintosh MIDI Manager which then relays commands to sound generating devices attached to the computer. Using MIDI gives us the flexibility which we believe is essential for auralizing arbitrary programs. The user can generate simple musical sounds through a primitive synthesizer responding to MIDI commands or sophisticated sampled sounds by connecting a different device. Separating the sound generation from the programming environment using MIDI has reduced the possibility that the auralization itself will cause problems in a program being debugged.

LogoMedia has three new commands for generating sound: `startnote`, `playnote`, and `stopallnotes`. `startnote` begins playing a MIDI sound of a given channel, note number, and volume. To turn off a note, the volume of zero is used. The `playnote` command has the additional parameter of duration specified in 60ths of a second. `stopallnotes` turns off sound in all 16 MIDI channels.

Specifying the monitoring of execution values

In order to facilitate the rapid association of sound with execution behavior we have devised a tool for mapping expressions to sound qualities. The audio expression tool tells LogoMedia what sounds to generate during execution based on changes in the program information. Each line in the table describes how an expression containing one or more program variables is related to particular sound qualities of a specified voice. The tool as depicted in Figure 5 was designed to encourage users to use sound in ways which have been successful in previous auralization applications. Specifically, the audio expressions tool makes it easy to differentiate by timbre a collection of runtime values being monitored and discourages users from mapping this data to more than one audio dimension.

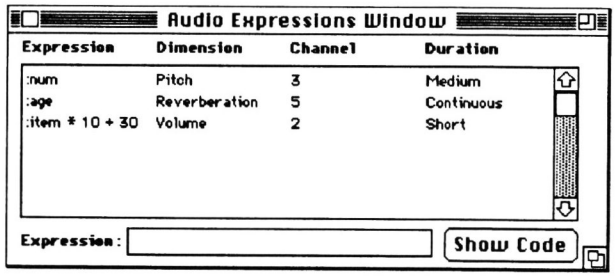

Figure 5: *LogoMedia's expression-sound table for observing changes in program variables*

To prepare a runtime auralization using the audio expression tool the programmer enters a Logo expression containing variables used in his or her code. We chose to map sounds to expressions instead of simply variables to allow the user to scale and offset their values to fit the range expected by the MIDI devices. Pitch, for instance, is specified by MIDI with a value between 0 and 127, yet many instruments, samplers in particular, cannot play notes which cover this entire range. After entering the expression programmer then selects an audio dimension to which the value of the expression will be mapped. Currently, the only available dimensions are pitch and volume, although we plan on adding the ability to control reverberation, stereo panning, and envelope. The next step for the programmer is to select on of 16 MIDI channels which determines the timbre of the resulting auralization for the expression. Lastly, the programmer specifies the duration of the sound as either "continuous" or one of three relative time periods—short, medium, and long. The continuous selection implies after the

[1]Invasive is the term used by Price, Small, and Baecker (1991) to categorize software visualization systems which require modification to the source code. It seems an appropriate descriptor for program auralization systems as well.

initialization of any variable in the expression the MIDI channel will play until the program terminates. The relative periods have varying duration depending on the rate of execution selected by the user.

Because of the ease of creating meaningless cacophony, the audio expression tool was deliberately designed to constrain the user to auralization techniques that show the most promise. The layout of the window, for instance, encourages users to differentiate their expressions by timbre and discourages mapping its value to different sound channels. Additionally, the design of the tool makes it difficult to monitor the same expression using more then one audio dimension. A more flexible interface which allowed expressions to be entered for each sound quality including timbre was rejected, since it provided no guidance as to the more effective uses of sound. We have also rejected the ability to enter absolute duration times, since this reduces the adaptability of the auralization to various rates. We believe that the capability to play back programs at different speeds we believe is key to deriving meaning from auralizations and this is planned to be a focus of our user study.

The constraints imposed by the audio expressions tool reflect simply techniques which have been effective in the past, but we cannot expect the tool to facilitate all useful "sonifications." However, the audio expressions tool can be used to generate a first approximation of the appropriate auralization which can later be refined by the programmer using the Logo language. By pressing the "show code" button the user can see the underlying Logo code responsible for auralizing a particular Logo expression. This code can then be copied into a LogoMedia document and edited.

Informal observations from the use of the audio expressions tool reveal weaknesses which must be addressed in the next iteration of the interface before user testing. Scaling and offsetting a variable to fit MIDI protocol has proved cumbersome. A more fundamental problem is predicting the range of values the variable will take on. This was a challenging task even for the original author of some simple turtle drawing programs. To expect the programmer to know the range of a variable is in a sense begging the question: if the programmer knew its variance, there might be no point to auralizing. This problem is not unknown to the field of software visualization in which limits are imposed by the boundary of screen coordinates. Our next version of LogoMedia will monitor the range of values during each execution of a program and present this information to the user to help guide the next iteration of the auralization specification.

Specifying the monitoring of execution events

In order to provide LogoMedia with simple non-invasive techniques for auralizing control flow, we are developing a method for associating audio commands called audiopoints with individual lines of code. Figure 6 illustrates a sample LogoMedia editor window with the Logo code on the right and its associated audiopoints on the left. To add an audiopoint to a line of the program the user simply positions the cursor in the left column beside the desired line and types a Logo statement. Alternately, the programmer after moving the cursor over the appropriate column can simply play a note on an attached MIDI keyboard. The corresponding LogoMedia command for reproducing this note will then appear as an audiopoint. As the program is running LogoMedia will execute each audiopoint immediately prior to the line of code on its right.

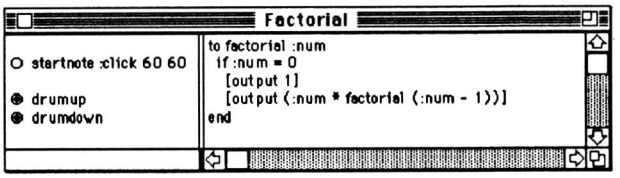

Figure 6: *LogoMedia's two column format for associating audio with program events*

A simple audiopoint such as on the second line can generate a fixed length tone as the code is executed to indicate the line has been reached. Additionally, the programmer can call his or her own custom auralization procedures or choose from a library of useful sound procedures. We are constructing such a library with the ability to toggle sounds on and off and layer sounds. The audiopoint on line 4, for instance, causes drumming sounds to become increasingly dense as the factorial program descends into successively deeper levels of recursion. As the program pops out of the nested levels, the audiopoint on line 5 causes the sound to become less dense.

Conclusions

Whether sound can be used effectively in programming environments remains to be determined. Testing and evaluation of LogoMedia is planned to ascertain which applications of audio in the program development process are most useful. We do not expect sound to replace either traditional textual representations or software visualizations, but to complement them. Further research is needed to determine how all three modalities can be best synthesized to take advantage of the multi-media capabilities of today's computers. We hope that by enhancing the development environment with sound we can make programming more engaging and empower the programmer with tools for more accurate entering, faster debugging, and an improved understanding of a program's function, structure, and behavior.

Acknowledgements

We appreciate the support to our laboratory from the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Excellence of Ontario, and Apple Computer, Inc.

References

Baecker, Ronald M. (1981). *Sorting Out Sorting*. Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto. 16 mm color sound film, 30 minutes, distributed by Morgan Kaufman Publishers.

Baecker, Ronald M., and Aaron Marcus (1990). *Human Factors and Typography for More Readable Programs*. Reading, Massachusetts: Addison-Wesley.

Baecker, Ronald M., and J.W. Buchanan (1990). A Programmer's Interface: A Visually Enhanced and Animated Programming Environment. *Proceedings of the Twenty-Third Annual Hawaii International Conference on Systems Sciences*, 531-540.

Blattner, M., D. Sumikawa, and R. Greenberg (1989). Earcons and icons: Their structure and common design principles. HCI 4 (1): 23-37.

Bly, Sara (1982). Presenting information in sound. *Proceedings of the CHI '82 Conference on Human Factors in Computer Systems*, 371-375.

Brown, Marc H. (1988). Exploring Algorithms Using Balsa II. *IEEE Computer* 21 (5): 14-36.

Brown, Marc H., and John Hershberger (1991). *Color and Sound in Algorithm Animation*. Digital Equipment Corporation. Systems Research Center Report, 76a.

Buxton, William,William Gaver, and Sara Bly (in preparation). *Auditory Interfaces: The Use of Nonspeech Audio at the Interface*. Cambridge University Press.

Eisenstadt, M., and M. Brayshaw (1988). The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming* 5 (4): 1-66.

Gaver, W. W. (1989). The Sonic Finder: An interface that uses auditory icons. *HCI* 4 (1): 67-94.

Gaver, W. W. (1990). Auditory icons in large-scale collaborative environments. *Proceedings of the Human-Computer Interaction - Interact '90 Conference*,

Gaver, W. W. (1991a). Effective sounds in complex systems: The ARKola simulation. *Proceedings of the ACM Special Interest Group in Computer-Human Interaction Conference*, 85-90.

Gaver, W. W. (1991b). Sound Support for Collaboration. *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, 293-308.

Gellenbeck, Edward M., and Curtis R. Cook (1991). Does Signaling Help Professional Programmers Read and Understand Computer Programs? *Proceedings of the Empirical Studies of Programmers: Fourth Workshop*, 82-98.

Harvey, B. (1986). *Computer Science Logo Style: Intermediate Programming*. Cambridge: MIT Press.

Henry, R. R.,K.M. Whaley, and B. Forstall (1990). The University of Washington Illustrating Compiler. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 223-233.

Hutchins, E. L., J. D. Hollan, and D. A. Norman (1986). Direct manipulation interfaces. In *User centered system design: New perspectives on human-computer interaction*. Edited by D. A. Norman and S. W. Draper. 87-124. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

IMA (1983). *MIDI musical instrument digital interface specification 1.0*. North Hollywood, CA: IMA. (Available from IMA, 11857 Hartsook Street, North Hollywood, CA, 91607, USA)

Jackson, Jay Alan, and Joan M. Francioni (1992). Aural Signatures of Parallel Programs. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 218-229.

Jenkins, James J. (1985). Acoustic information for objects, places and events. *Proceedings of the First International Conference on Event Perception.*,

Kesler, T. E., and R. B. Uram (1984). The effect of indentation on program comprehension. 21 415-428.

Mezrich, J. J.,S. Frysinger, and R. Slivjanovski (1984). Dynamic representation of multivariate time series data. *Journal of the American Statistical Association* 79: 34-40.

Nanja, Murthi, and Curtis R. Cook (1987). An Analysis of the On-Line Debugging Process. *Proceedings of the Empirical Studies of Programmers: Second Workshop*, 172-184.

Patterson, R. D. (1989). Guidelines of the design of auditory warning sounds. *Proceedings of the Institute of Acoustics 1989 Spring Conference*, 17-24.

Price, Blaine A.,Ian S. Small, and Ronald M. Baecker (1992). A Taxonomy of Software Visualization. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 597-606.

Small, Ian S. (1989). Program Visualization: Static Typographic Visualization in an Interactive Environment. Masters Thesis, Department of Computer Science, University of Toronto.

Smith, Ben (1991). Unix Goes Indigo. *Byte* 16 (9): 40-41.

Smith, Stuart,R. Daniel Bergeron, and Georges G. Grinstein (1990). Stereophonic and Surface Sound Generation for Exploratory Data Analysis. *Proceedings of the CHI'90, ACM Conference on Human Factors of Computing Systems*, 125-132.

Sonnenwald, Diane H.,B. Gopinath,Gary O. Haberman,William M. Keese III, and John S. Meyers (1990). InfoSound: An Audio Aid to Program Comprehension. *Proceedings of the Twenty-Third Hawaii International Conference on System Sciences*, 541-546.

Speeth, R.D. (1961). Seismometer sounds. *The Journal of the Acoustical Society of America* 33 (7): 909-916.

Weinberg, Gerald (1971). *Psychology of Computer Programming*. New York: Van Nostrand Reinhold Company.

Yager, Tom (1991). The Littlest SPARC. *Byte* 16 (2): 169-174.

Yeung, E.S. (1980). Pattern recognition by audio representation of multivariate analytical data. *Analytical Chemistry* 52 (7): 1120-1123.