FAUST: A Framework for Algorithm Understanding and Sonification Testing

Jessica R. Weinstein
Silicon Graphics, Inc.
jrw@sgi.com

Perry R. Cook
Princeton University
prc@cs.princeton.edu

ABSTRACT
One of the main obstacles to experimentation with algorithm sonification is the lack of simple tools for auditory display. There is considerable overhead both in creating sounds and in deciding which sounds are best suited to a particular algorithm. The goal of FAUST is to provide a framework that programmers can use to easily sonify their programs. FAUST is a modular tool which allows simple mapping of algorithm events to sound parameters and provides the actual structure needed to create the sound. Programmers can thus test sonification options without having to understand the underlying sound synthesis mechanisms. However, the underlying sound synthesis framework does allow interested programmers to easily change sound synthesis algorithms, or to add features and parameters to the existing algorithms.

Keywords
algorithm sonification, data representation, auditory display tool

INTRODUCTION
It is often said that many of the greatest innovations in the field of computer science have been in understanding the way humans use computers. This understanding has allowed computer designers and programmers to create a much more efficient and flexible environment for computer users - from machines with simple blinking lights and switches we quickly progressed to powerful yet intuitive graphical interfaces. However, until very recently the interaction between humans and computers has been severely limited by the fact that all information provided by the computer had to be graphical or textual in nature. Whereas humans are able to use all five senses in our interaction with the world around them, computers only made use of our vision. In the last ten years we have finally begun to see some progress in this area. It is our hope that we can use this technology to help programmers in their attempts to understand, debug or explain algorithms.

Auditory Display
When it comes to using sound, there are a wide variety of different types of sound computers can produce. These include music, speech, sounds that mimic real-world noises, and auditory displays. Each type of sound is useful given certain needs and criteria. Music is useful because it guides our emotional response to visuals and provides vitality and action. It can be harmonious or discordant, familiar and reassuring or new and unexpected. Speech is often used for detailed and specific information. It generally requires concentration to understand, and cannot be usefully combined with other complex information. Real-world sounds mimic the natural sounds of the world around us. This approach takes advantage of our life-long acquaintance with these sounds, meaning they come with a built-in context and meaning and evoke familiar responses.

Auditory display is the representation of data through non-speech sound. Here we can map parameters of visual images into parameters of sound, or map parameters of the data directly to audio parameters.

The focus of this project is on the specific area of using sound to elucidate algorithms, so auditory display seems most suitable to the needs of such a system. There is no real need to take advantage of the real-world context accompanying music, speech, and real-world sounds. In fact, a more abstract representation seems in order when we are representing arbitrary algorithms; any real-world context associated with the sounds could be confusing and counter-productive.

Motivation

Like Brown and Hershberger [3], we see four main reasons for the use of sound in algorithm animation. First, audio cues can be used to reinforce visual views. Auditory information that is redundant with visual information allows us to exploit the strengths of each mode and to provide cues about events that might not otherwise be visually attended. Second, sound is particularly well suited to conveying patterns and relationships in data that would not be seen or noticed in visual displays. This is because the ears are excellent at certain types of data analysis, like picking up correlations and repetitions or identifying nonlinear sequences of variable values. For example, it might be difficult to find an infinite loop in a piece of code by looking at it, but listening to the algorithm might make it quite easy. Third, an excellent property of sound is that it can convey information that is especially difficult or awkward to display graphically. It allows simultaneous representation of multiple variables without confusion. Auditory displays are also able to reduce visual clutter by providing an alternative information source. Audio can thus provide users with debugging or profiling information, for example, without disturbing the integrity of the graphical interface. Finally, audio cues can help identify important events in a stream of data. Sound has an insistent quality that works especially well for alarming the user about an occurrence that needs to be dealt with immediately.

There is, however, a drawback to the use of sound in understanding algorithms. Whereas graphical displays are fairly well understood, sound is difficult to use effectively. One of the obstacles to widespread experimentation with algorithm sonification is the lack of a tools allowing programmers to sonify an algorithm easily. There is considerable overhead involved both in creating sounds and in deciding which sounds are best suited to a particular algorithm. The goal of this project is thus to provide a framework that programmers can use to easily sonify their programs. We have named this system FAUST: a Framework for Algorithm Understanding and Sonification Testing. FAUST provides intuitive capability to map algorithm events to sound parameters, as well as the actual structure needed to create the sound. In this way, the programmer can experiment with auditory display and test different options without having to deal with the actual sound synthesis mechanism.

BACKGROUND

The idea of using non-speech audio to aid computer use and scientific visualization is rather new, and largely due to advances in workstation technology which made it easy to generate sound. Until the last half of the 1980s, the computer world was largely uninterested in auditory data representation, due to the difficulty and expense of providing sound hardware and software in the platforms then available to the programmers. This was changed by the development of the Musical Instrument Digital Interface (MIDI), which made computer control of sound synthesis orders of magnitude easier for programmers. In addition, personal computers began to be

shipped with multimedia hardware allowing ordinary computer users access to high-quality playback of sounds. However, although sound has seen limited use in some software visualization systems, serious applications of computer audio to the programming domain have been limited until about the last five years.

Auditory User Interfaces

Most early use of sound to aid computer use was focused on enhancing the computer's user interface. This problem was broached by a number of researchers, most notably by Gaver [7]. Gaver created SonicFinder, an audio-enhanced version of the Macintosh Finder application, in 1989. SonicFinder tagged common operations or events with everyday sounds to help identify and call attention to them. Examples included audio cues which indicated the size of a file through pitch and crash sounds when the user dragged a file to the trash. A less complete version of this type of sonically enhanced interface was soon incorporated into mainstream graphical interfaces such as Silicon Graphics' IndigoMagic Desktop.

These types of event-driven sounds were later dubbed "earcons" by Blattner [1], who suggested that properly designed short melodies could be learned quickly and associated with arbitrary events. However, Blattner pointed out the major drawbacks of real-world sounds were the very realism of the sounds, which sometimes carried a larger context than the programmer wished. She used representational auditory icons which were sometimes based on real-world sounds, but did not exactly correspond to real-world behaviors. Blattner and researchers who followed developed a syntax for auditory messages - design principles that could be used to create messages in an auditory interface.

Data and Program Visualization

At the same time, researchers such as Smith, Bergeron, and Grinstein [8] were attempting to gauge the possibilities of applying sound to the problem of visualizing complex data. One key application was in the visualization of multi-dimensional data, where another channel was needed to convey the information not visible on a computer screen. More recent research has begun to demonstrate the potential of non-speech sounds in the programming domain. Brown and Hershberger created a seminal work which used non-speech audio to graphically and sonically display hash tables [3]. Their system generated sounds corresponding in pitch to elements being inserted into the hash table and items being sorted; additional sounds were used to identify when a hash collision occurred. Neither of these works replaced graphical animation with sound; instead they used sound to enhance and complement information already represented.

Jackson and Francioni [6] then created some of the first applications which used sound as their primary medium for visualizing the state of an algorithm. Their system used audio to help users understand the behavior of parallel algorithms by generating sounds based on trace data recorded during execution. Each processor was mapped to a different timbre, and varying pitches denoted the various events. Using sound to elucidate parallelism proved to be a worthy area of research, but in most research efforts the sonification of the algorithm was a post-processing step which depended on trace data generated by the algorithm during execution.

It is only in the last five years that serious research has been done on the auditory display of arbitrary algorithms, rather than specific applications. This project attempts to create a very general framework where sound can be applied to a highly varied programming environment.

This potentially allows programmers to use sound to elucidate (and possibly debug) the behavior of any algorithm, with or without the additional use of graphical visualization systems. Indeed, FAUST does not even require that its input is an algorithm - it merely requires a stream of tagged events. This will be discussed in more detail in the following sections.

DESIGN

Auditory display of arbitrary algorithms is a challenging interface design problem because the sounds must be adaptable to a wide variety of algorithms and conditions. In addition, we would like to be able to represent both data and events simultaneously, in a user-defined fashion. An important distinction to make before we begin discussing the design of FAUST 1.0 is that this project does not propose to dictate which sound mappings will help the programmer understand programs easily. The point of this research is not so much to generate intuitive hints and suggestions, as to create an environment where other researchers can easily determine which mappings work best for particular algorithms and types of events.

The FAUST framework design consists of a very simple architecture, with three interlocking parts: the algorithm module, the mapping workspace, and the instrument module. The interaction between these parts is shown in figure 1. The framework is specifically designed to be transparent to the user, so that the programmer has to know as little as possible about the mechanism involved in the auditory display. Minimizing the amount of input required from the user was an important priority - ideally, the user should have to know only a minimal amount about the system in order to use it effectively. The system is also designed to be modular enough to be easily extensible by the user at all layers, beginning with the input event tagging, through the mapping workspace, and down into the final sound synthesis algorithm.

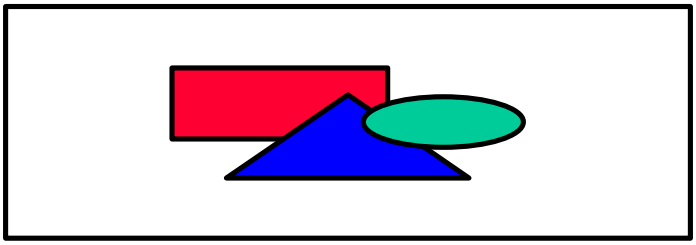

Figure 1: The FAUST architecture

The Algorithm Module

Programmers who wish to sonify their code using the FAUST framework would begin by choosing which events in their algorithm should correspond to auditory events. Examples of this include the swapping of two elements in a sorting algorithm, and the traversal of an edge in a graph algorithm. Each event would then be given a unique name, and a list of these names stored in a small text file.

The programmer would then link the algorithm to be sonified with the FAUST algorithm module. This module is a small set of C functions which provide the structure to communicate with the mapping workspace without forcing the programmer to know anything about the underlying communication between modules. The code includes an initialization function, as well as a function designed to tag important events. That is, whenever one of the earlier specified events occurs, the programmer calls this function to notify the mapping workspace. For example, if this were a sorting algorithm, the programmer would probably want to use this tagging

function to notify the mapping workspace each time there was a swapping of two elements. This module is purposely kept small to ensure that it can be easily modified by the programmer if desired. Indeed, the programmer could create an entirely different API to FAUST simply by writing a new algorithm module.

The Mapping Workspace

The Mapping workspace module acts as a graphical patchbox: the programmer begins by choosing an instrument from a list of instruments, ranging from abstract waveforms to physical models like bells and chimes. The instrument choice can be changed at any time, even while the algorithm is running. A list of the previously specified algorithm events is displayed in a column on the left of the patchbox. Sound parameters such as pitch, brightness, spatial location, etc. are displayed in a column on the right.

The user can now map algorithm parameters to sound parameters by using the mouse in the patchbox window display - left mouse clicks will create mappings, middle mouse clicks will delete them. When the algorithm is run, input values are mapped to output according to the mapping links in effect at the time. These can be changed at any time while the algorithm is running to help the programmer find the most effective mapping for a particular program.

When users find a successful set of mappings, they can save the set of mappings and instrument choice and store it for later use. The idea behind this is to let a user experiment with a variety of different mappings and save the one that works best with a particular algorithm. All mappings saved in this manner are stored away in a file, and can later be reloaded for further experimentation. Of course, these mappings can then be modified at any time after they are loaded.

The mapping workspace is implemented in Tcl/Tk, a scripting language and set of widgets which have been implemented on all major platforms. This gives FAUST cross-platform capability by allowing the mapping workspace to be run under almost any operating system. All initial testing of FAUST was performed on a Silicon Graphics machine, in an X-Windows environment.

The Instrument Module

The mapping workspace in turn communicates with the instrument module. Each time the mapping workspace receives a message from the algorithm, it looks up the mapping for that type of event. If one exists, the mapping workspace transmits a corresponding audio message to the instrument module. These messages map the events to a variety of audio parameters. These primitive sound parameters can be combined to create complex commands, such as playing musical patterns. The sound parameters used in FAUST 1.0 are based on the capabilities of the underlying Synthesis ToolKit and the desire to create simple mappings, and include pitch, signal/noise, note duration, brightness or timbre, and spatial location.

The instrument module thus includes both the Synthesis ToolKit and the instrument server, a small front-end which receives messages. The Synthesis ToolKit [4] is a collection of C++ classes which can quickly create and connect real-time music synthesis and audio processing systems. The ToolKit includes a variety of synthesis algorithms which share a common interface. The individual algorithms can be combined to create real-world-sounding instruments, ensembles of instruments, and sound effects. FAUST uses a smaller subset of these algorithms to create a few simple instrument possibilities for the user. Each instrument is based on a

combination of "UnitGenerators" - classic building blocks in computer music. These UnitGenerators include oscillators, filters, delay lines, etc. The UnitGenerators can be optimized individually as needed to quickly create efficient instruments.

The default output instrument is called `Simple`, and it is designed to be just that. `Simple` is an abstract harmonic waveform with additive noise, and is not designed to mimic any physically based instrument. This avoids the problem of real-world context interfering with the auditory display. The parameters to this instrument, as previously mentioned, are pitch, noise, duration, brightness and location.

The instrument's pitch is directly controlled by an oscillator. The noise component comes from a random number generator, run through a tuned noise filter. This ensures that even a completely noisy signal has some intrinsic pitch. The outputs of the oscillator and the tuned noise filter are added, and the resulting sound is run through a brightness filter, which gives a simple control over the overall timbre of the sound. The sound is also modified by an envelope generator, which determines the time-amplitude relationship (and duration) of the sound. Finally, a parameter was added to the instrument which controls its spatial location. The instrument is played in two channels, left and right, and a variable `pan` is assigned values between 0 and 1 to control what proportion of the sound is played from each channel.

However, because real-world context is sometimes desired, the initial version of FAUST also includes three other instruments. These instruments are similar to the Simple instrument, except that they are spectral and physical models of real-world instruments. These include a chime, a muted cowbell, and a plucked guitar string model. Each uses the same five parameters as the `Simple` instrument. The Synthesis ToolKit currently includes over two dozen instruments, with more being added. The common interface means that these instruments can easily be added to FAUST at any time.

The input to the Synthesis ToolKit is in the form of messages received from the mapping workspace and passed on by the instrument server. These messages use an extended MIDI language called SKINI (Synthesis toolKit Instrument Network Interface) [4]. SKINI messages are a protocol similar to MIDI, and they are designed to be compatible whenever possible. One large difference between the two protocols is that SKINI employs a text-based control stream, with meaningful, easily tokenizable names. This means that any system that can transform strings into strings, floats and ints can parse and process SKINI messages. By the same token, any system capable of printing is able to generate SKINI messages. Sequences of SKINI messages can be saved as scores, and then easily edited, searched or debugged by humans. Secondly, SKINI, unlike MIDI, allows for double precision floating point accuracy in note numbers, control values, velocities, timing variables, etc. This extra precision can be ignored to create fully MIDI-compatible byte values.

EXPERIMENTS

In the experiments described here, we were looking for two things which we felt would reflect the effectiveness of this initial version of FAUST. The primary concern is how easily programmers are able to sonify their algorithms. As detailed in the previous sections, a large part of the design of FAUST was focused on making this process as simple as possible. It would certainly be interesting to discover how successfully the current implementation meets this goal. A secondary concern is how effective the sonification is in helping users to understand the actual

algorithms being sonified. It is important to remember that this is the core motivation for the existence of a system like FAUST. If sonification does not actually help us to understand algorithms, there is no point in creating a framework to make it easier to implement.

The main application used in our experiments is a program called "Sorting In Action Noisily", a version of which was originally developed as a demo for the NeXT platform by David Jaffe of NeXT, Inc. This version was written for the SGI platform and includes graphics capabilities using the OpenGL graphics library. Sorting In Action Noisily is a simple integer sorting program which provides five standard sorts from which the user can choose. These include bubble sort, insertion sort, selection sort, shell sort and quicksort. For the purposes of the experiment, the sorting algorithms were labeled "Sort1" through "Sort5". This made it possible to test whether the subjects could identify the current sort algorithm just by listening to it.

The experiment was conducted as follows: the subjects were told to track swapping events during the sort. That is, we would like to hear some change in the audio output each time two elements in the array of integers are swapped. We do this by specifying two events - `SwapValue`, which would reflect the value of each element being swapped, and `SwapIndex`, which would correspond to the array index of the element being swapped. All subjects, given the complete sorting program, were able to correctly implement this configuration using the algorithm module and a small amount of documentation. The subjects were accomplished programmers familiar with the Unix environment, but these are exactly the types of people expected to be able to make the most use of a system like FAUST. The fact that they were able to easily sonify all five sorting algorithms using the given events is encouraging.

The subjects were then asked to run FAUST using their new events and to spend a few moments testing out the mapping workspace. They immediately saw how to create mappings, although it was not always obvious how to clear them once created. After running a few sorts, every subject chose pitch as the most intuitive mapping, and chose to use this sound parameter exclusively during the rest of the experiment. We then asked each subject to listen to each sort as many times as they wished, and to attempt to identify which set of sounds corresponded to which sorting algorithm. During this time no graphical output was displayed. To make this a bit easier, the subjects were given the names of the fives sorts represented. The subjects were familiar with all five algorithms. This experiment was repeated four times, twice using only the `SwapValue` event and twice with only the `SwapIndex` event. Each time the order of the algorithms was changed.

The subjects claimed to rely heavily on their knowledge of the sorting algorithms to help them identify the algorithms. In particular they mentioned the relative speed of the algorithms and the proximity of the values being swapped. However, on average, they were only able to correctly identify three of five sorts, and there was no significant difference between the `SwapValue` and the `SwapIndex` runs. Some subjects did identify all five sorts using `SwapValue`. However, when even these subjects were asked to repeat the experiment using the other four sound parameters, none of them could clearly differentiate any of the sorting methods. Using `SwapIndex`, `SwapValue` or both simultaneously did not seem to affect the outcome. This is consistent with knowledge of music perception. Other parameters could be added which are more perceptually salient, and some of the existing parameters could have more meaning in a different algorithm context.

The final stage of the experiment used graphical output only, with no audio mappings. When the

experiment was repeated graphically, almost all subjects were able to identify the correct sorting procedures with complete accuracy. It would seem that the auditory display is still inferior to its graphical counterpart - what is not clear is why. One interpretation is that the graphical display is simply more familiar; it is possible that the subjects would be able to better understand the auditory display given more exposure to a variety of algorithm sonifications. Given more time, it would be interesting and enlightening to repeat this experiment with algorithms that could not easily be displayed graphically and compare the results. Experiments displaying parallel algorithms or complex data suggest that the outcome might be dramatically different [6].

RELATED WORK

In recent years, a number of researchers have been creating frameworks which involve the use of sound to understand algorithms. We feel that all of these systems are fundamentally different from FAUST. One such system is LogoMedia for the Macintosh [5], a sound-enhanced debugger written by DiGiano and Baecker. LogoMedia uses both symbolic and iconic sounds to tag events in the algorithm. For example, users can play a bird "tweet" noise every time a particular statement is executed or hear a cymbal crash every time a variable's value is modified. FAUST differs from LogoMedia in that FAUST users do not need to memorize the meanings of a complex and arbitrary set of bird tweets, dog barks and crash noises. Instead, FAUST uses simple parameters of sound such as pitch and spatial location, and lets the user choose which parameters designate which algorithm events. The framework is open-ended and general enough to allow new mappings to be added easily. A graphical display helps to remind programmers which mappings they have chosen.

Another related system is Jameson's Sonnet [9], an audio-enhanced debugger based on his own visual programming language. Sonnet has the very useful capability of non-invasive event-linking: whereas FAUST users must manually tag events in their algorithms, Sonnet will automatically identify potential candidates for sonification. This could potentially be a useful addition to future versions of FAUST. However, FAUST's design also differs from Sonnet's in that Sonnet runs embedded in a specific debugging environment. FAUST is not dependent on any particular debugger or application, or even on any particular platform.

A third related work is CAITLIN [11], by Vickers and Alty. CAITLIN differs from FAUST in that it produces musical output. FAUST produces generic sound, and could only be said to produce music in any sense if the algorithm it is linked to generates "musically interesting" parameters. In addition, CAITLIN's sound parameters are part of its pre-processing step, and as such there is no counterpart to FAUST's mapping workspace, and no interaction with the user. Some related systems use their own languages to specify program auralization. These include the Auditory Domain Specification Language (ADSL) [2] and the Listen Specification Language (LSL) [10]. The final input of the CAITLIN, ADSL and LSL preprocessors is in the form of an executable which must be regenerated each time there is a change of source code or sound parameter preference. FAUST, on the other hand, processes algorithm events in real time, meaning the algorithm doesn't need to run to completion each time the source is changed for a new stream of events to be produced.

A final related system is a data sonification toolkit called Listen, by Wilson and Lodha [12]. Listen is similar to FAUST in its intent to provide a "flexible, extensible, portable, and interactive toolkit" to encourage sonification research, but it differs from FAUST in several key areas. Most importantly, their system is based primarily on MIDI, which is fixed (by design), has

very few parameters except for volume, pitch, and panning, and is not really extensible. FAUST, on the other hand, uses the Synthesis ToolKit, which allows the programmer to arbitrarily add new sounds and parameters to the system through its flexible source-code-available synthesis algorithms.

Finally, we feel there are two more advantages to FAUST's design over other conventional sonification systems. Although parts of this system can be found in the other sound-based research, we are not aware of any system that combines them all into one framework. The first advantage is FAUST's modularity. Each of the three individual components of FAUST is implemented as an entirely separate process from the other two. The primary mode of communication between the three components is via sockets. This means that the three processes can run independently on any platform that supports sockets. For example, the mapping workspace could be running on a Windows NT machine, the instrument module could be running on a Silicon Graphics workstation, and the algorithm could be running on a Linux box. Implementing the architecture in this manner gives the user the freedom to utilize the best aspects of each platform. In addition, it allows modular changes to be made to the system without requiring changes in the other components.

A final strength of the project, and probably its most compelling and unique one, is FAUST's flexibility. This is largely due to the underlying source-code-available Synthesis ToolKit. Programmers who wish to use different sounds, or create new sounds of their own, can easily swap out the current sound synthesis algorithms for new ones. This flexibility extends beyond the sound synthesis to the other components as well. The algorithm module is highly extensible, allowing users to customize the method of input tagging easily. This allows FAUST to sonify any type of algorithm or data set, using a user-specified API if desired. In addition, the compact size of the algorithm module and the fact that the Synthesis ToolKit is a set of C++ classes combine to allow the programmers to extract the parts of FAUST that are most useful to them. These parts could then be embedded permanently into an application, demo, pedagogical example, or other larger work. Hence FAUST is a unique programmers sonification toolkit, not an audio component of a debugger.

CONCLUSION AND FUTURE WORK

It is clear that there is a need for more experimentation. The only way to make this framework truly useful to programmers is to ask as many of them as possible to test it with their algorithms. Their feedback would be useful in determining how to make future versions of FAUST easier to use, what additional mapping functionality might be necessary, and how to tweak the synthesis algorithms to provide the most effective sound parameters. In addition, programmers could provide suggestions as to what kinds of mappings tend to be most effective in aiding algorithm understanding, and indeed what types of algorithms work best with an auditory display. Information gleaned from good auditory displays could be used both to further understanding of current algorithms, and to help in the development of new and better ones.

One difficult problem in implementing FAUST 1.0 was trying to synchronize the auditory display with the algorithm being displayed. The two sockets connecting the components of FAUST, while allowing a lot of flexibility, also represent a significant amount of overhead. Synchronization is not a major issue in all cases: the time delay is relatively small, and if the programmer is testing an algorithm with no graphical display, it will probably be impossible to tell that the auditory response is not instantaneous. On the other hand, algorithms with a

concurrent visual display are expected to have the sound and visual streams convincingly synchronized in order to avoid confusion. If the two media are not correctly synchronized, many of the advantages of using sound disappear. Satisfying these synchronization needs is unfortunately largely dependent on the operating system. We attempted to reduce the delay between the visual and auditory displays as much as possible by minimizing buffering within the operating system and making the Synthesis ToolKit as efficient as possible. SKINI and the Synthesis ToolKit allow for absolute and relative timestamps of events, and this mechanism could be used to ensure synchronization between graphics and sound. This is certainly an area for future work, both on the part of FAUST and inside future operating systems.

In addition, we feel that the mapping workspace could be greatly improved by allowing more flexibility and mapping choices. For example, it might be useful to allow programmers to scale and combine sound parameters and map them to one algorithm event, or vice-versa. This would allow programmers to specify, for instance, the pitch of the output sound to be controlled by both Event1 and Event2, with Event1 contributing 60% of the value and Event2 the remaining 40%. Unfortunately this is currently an unsolved problem, especially from an interface standpoint. It is not clear how to provide an interface that gives users this capability without simply confusing them and rendering the whole interface (and thus other capabilities) complicated to the point where it is unusable.

Finally, a shortcoming of this initial version of FAUST is that it requires programmers to manually tag events in their algorithms. For many algorithms there are "obvious" events that can be tagged, such as the now familiar swapping events in a sorting algorithm. It would make FAUST more effective and easier to use if this process could be at least partially automated. Ideally, the sonification framework would choose some promising events in each algorithm and let the user modify them or specify additional events.

Although the initial results of experiments using FAUST 1.0 to sonify algorithms were promising, there are a lot of opportunities for future work. Indeed, auditory display is a field of computer science which is currently so poorly understood that it may take years before we are capable of fully exploiting this underutilized medium. It is our hope that the creation of FAUST will help to remove some of the obstacles to experimentation with algorithm sonification and allow programmers to explore new possibilities and directions.

REFERENCES

1. Blattner, M.M., Sumikawa, D.A., and Greenberg, R.M. Earcons and Icons: Their structure and common design principles. *Human-Computer Interaction*. 4, 1 (January 1989), 11-44.

2. Bock, D.S. ADSL: An auditory domain specification language for program auralization. In *Proceedings of ICAD '94*. ICAD, Santa Fe, NM, 1994.

3. Brown, M.H., and Hershberger, J. Color and sound in algorithm animation. *IEEE Computer*. 25, 12 (December 1992), 52-63.

4. Cook, P.R., Barger, R., Serra, X., and Freed, A. SIGGRAPH Course Notes # 17 and 18: Creating and manipulating sound and music to enhance computer graphics. ACM SIGGRAPH, New Orleans, LA, 1996.

5. DiGiano, C.J., and Baecker, R.M. Program Auralization: Sound enhancements to the programming environment. In *Proceedings of Graphics Interface '92*, 44-52. Canadian Human Computer Communications Society, Vancouver, Canada, 1992.

6. Francioni, J.F., Albright, L., and Jackson, J.A. Debugging parallel programs using sound. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 68-73. ACM SIGPLAN/SIGOPS, San Diego, CA, 1992.

7. Gaver, W.W. The SonicFinder: An interface that uses auditory icons. *Human-Computer Interaction*. 4, 1 (January 1989), 67-94.

8. Grinstein, G., and Smith, S. The perceptualization of scientific data. In *Proceedings of the SPIE/SPSE Conference on Electronic Imaging*, Vol. 1259, 190-199. SPIE, Santa Clara, CA, 1991.

9. Jameson, D.H. Sonnet: Audio-enhanced monitoring and debugging. In *Auditory Display: Sonification, Audification, and Auditory Interfaces*, G. Kramer, ed., SFI Studies in the Sciences of Complexity, Vol. XVIII, 253-265. Addison-Wesley, Reading, MA, 1994.

10. Mathur, A.P., Boardman, D.B., and Khandelwal, V. LSL: A specification language for program auralization. In *Proceedings of ICAD '94*. ICAD, Santa Fe, NM, 1994.

11. Vickers, P., and Alty J.L. CAITLIN: A musical program auralisation tool to assist novice programmers with debugging. On the World Wide Web at www.cms.livjm.ac.uk/www/homepage/cmspvick/caitlin/icad96.htm

12. Wilson, C.M., and Lodha, S.K. Listen: A data sonification toolkit. In *Proceedings of ICAD '96*. ICAD, Palo Alto, CA, 1996.