AudioPlusWidgets: Bringing Sound to Software Widgets and Interface Components

Benjamin K. Davison and Bruce N. Walker

Georgia Institute of Technology
Sonification Lab
654 Cherry St.
Atlanta, Georgia 30332-0170
ben@cc.gatech.edu, bruce.walker@psych.gatech.edu

ABSTRACT

Using sound as part of the user interface in a typical software application is still extremely rare, despite the technical capabilities of computers to support such usage. The ICAD community has developed several interface concepts, patterns, and toolkits, and yet the overall software scene has remained dominated by the visual-only user interface. AudioPlusWidgets is a software library offering scientifically grounded audio enhancements to the standard Java Swing API. Through metaphors and transparency, AudioPlusWidgets can be inserted into existing code with minimal changes, easily adding auditory capabilities to the interface components in the system. This library uses an event-based model and an audio manager to render speech, MIDI, and prerecorded sounds.

1. INTRODUCTION

Using sound to communicate information to the user of a software application is certainly not a new idea, especially within the Auditory Display community. As examples from the early days of ICAD, Mercator and Pink delivered entire operating systems with significant sound interfaces [1, 2]. However, sound in the user interface remains a very rare feature, despite decades, now, of interest in the topic. Part of the reason for this, we contend, is a lack of general-purpose, easily implemented, and empirically validated programming methods that allow developers to implement good auditory interface components quickly. This paper focuses on the development of a system of programmer-usable widgets (components of the interface such as buttons and scroll bars) that are enhanced with audio capabilities.

Unfortunately, years of auditory display research has only minimally impacted most software. While we are not aware of any comprehensive study on the use of sound in software, our searches and experience have revealed few applications in a typical home or business computer that use sounds beyond basic alerts and confirmations. We need to distinguish, here, the idea of adding audio to the actual interface controls, from the use of sound to present data, as in sonification or auditory graphs, and from the creation of sounds, per se. For example, there are certainly many sonification tools [3-7], and also programming tools such as JASS [8] and csound [9] that are used to create sounds, but they generally are not used to add audio to the user interface components. A notable exception is the class of (historically add-on) screen reader software, such as JAWS, [10] that adds simple speech output capabilities to an interface. Indeed, some operating systems now have speech output capabilities built in. But the use of any other kinds of audio enhancements (i.e., non-speech sounds) to the interface remains virtually nonexistent.

1.1. Why Are Audio Widgets Not Widespread?

One might argue that after twenty years of interest, perhaps the reason audio is not used more in interfaces is because it is not possible. Or perhaps it is not considered useful. Or perhaps it has simply not been implemented in quite the right manner. At risk of revealing our own intention, we subscribe to the last of these arguments, and for that reason have embarked on this project to develop audio enhancement tools that can be implemented by any programmer, with great ease and excellent results. But for the sake of argument, let us consider the other possible arguments.

A major concern of any invention is its usefulness in real situations. Does the lack of rich sound in software suggest that it is not useful? Certainly most software will work and even can be made generally usable without sound. While this question is a public concern among user interface software researchers, it is still largely unexplored and unanswered.

Some developers may consider that the usefulness or benefit of adding audio to an interface is small, relative to the cost of implementing it. Even if programmers believe in the benefit, the actual or perceived costs, in terms of implementation difficulties or simply extra programming, may tip the cost-benefit equation away from including audio. To rectify this situation, one can address the benefits (and perceived benefits) side of the equation, as well as the cost (programming difficulty) side of the equation.

On the benefits side of things, several researchers have demonstrated that audio can enhance a user's ability to complete tasks [11, 12]. In fact, the existence and vibrancy of the International Community for Auditory Display (ICAD) can be seen as a significant testament to the perceived value of using sound in this way. And of course, for persons with vision impairments, auditory enhancements have immediate and obvious benefits. This information, however, may be unknown to software engineers: A study by Lumsden confirms a general programmer ignorance about auditory displays [13].

However, it could be argued that programmers should not need to know or worry about the task benefits of audio enhancements in order to use them, so long as the costs are low. Software engineering, like much of engineering, is a highly modular process, dependent on outside groups to develop lower level parts. The programmer depends on software library developers to create the tools that are used to build software applications. For example, in programming a typical Java program, most of user interface development will focus on placing the widgets in the right place and making them call the right functionality. The Java library developers have already worked out how the button or slider should look and react in different operating environments. Programmers generally do not

question the usefulness of a slider, nor do they debate the colors or shapes or characteristics of the slider; they simply know it will allow the user to interact with the software in a predictable and effective manner.

While researchers developing auditory widgets in the past have demonstrated their work in a lab environment or in limited live environments, they have typically not sufficiently extended their code to be usable by the general software community. Further, each developer has implemented things in different ways, leading to a lack of standardization and no common or shareable tools. Given that each developer has had to make microscopic decisions about how to implement an audio-enhanced menu or button, it is no surprise that the implementation of auditory display research findings in these tools has been haphazard and project-specific.

This lack of a library hurts the auditory interface community as well. Since there is no standard or even competing standards, each lab must develop their process alone or go through the burden of integrating a colleague's code. Thus, one lab's implementation of earcons, for example, will almost certainly be different from that of another lab. This can lead to being unable to compare two works, and a fair amount of nearly redundant studies will have to be carried out. This constant reinvention of ideas greatly hinders the development of our science. If the structure of our science is instantiated in the structure of the software libraries we use, then what is currently known to be best practice needs to be obvious from the libraries' implementation. For the software developer, using these kinds of theoretically-derived and empirically-validated libraries means that his program is tuned to the state of the science. For the researcher, open avenues of research are clear since the traveled paths are exposed in the standard.

Thus, we come back around to the point that while there are many potential benefits for using audio in interfaces, for them to be realized we need to take another try at creating easily-implemented widget libraries that stand a better chance of becoming more widely adopted. This paper explains the development of such an audio-enhanced widget library for Java, called AudioPlusWidgets, or APWidgets for short. The APWidgets widget library extends many of the typical Java Swing widgets to include sounds. Sometimes, the sounds are nominal. In many cases, however, research concepts from the auditory display and auditory perception field have been implemented in novel and significant ways. Examples include advanced auditory menu elements, and soundscapes. The library depends on auditory concept metaphors but attempts to reduce the basic library interface to Java Swing metaphors. Like typical Java widgets, APWidgets can be implemented without any understanding on the part of the programmer as to why the "look and feel" (or sound) of the widget is as it is. The initial barriers to begin using sounds are dramatically reduced, although of course some understanding of the underlying theory will ultimately almost certainly lead to better interface designs.

The next section will briefly explore some of the great body of related work in this area. Section 3 describes the structure of the APWidgets library, including the motivation behind some of the design decisions. Section 4 provides some examples and comparisons between standard Java programming and using the APWidgets library. Finally, Section 5 wraps up this paper with a brief discussion on the current evaluation and efforts toward future work.

2. RELATED WORK

While developing this audio-enhanced Java widget library, we explored the related scientific work. In order to characterize the current state of the science, we explored the widgets invented by researchers over the years. These ideas, sometimes varying in schools of thought, provide the "way to make" the sound for the widgets. Following an object-oriented programming view of matching the code to the solution domain, we created classes that modeled the widget concepts. These ideas include earcons, auditory icons, spearcons, text-to-speech (TTS), and soundscapes. Other important ideas such as audio progress bars were explored but not implemented in the current version of the widgets.

The second avenue of research is the creation of usable Application Programmer Interfaces (APIs). A library is a set of code designed for a particular domain of functionality. An API is a programmer's front end to a given library. This includes objects, their properties, their methods, and author-created documentation about their use. For example, the Java Swing API is a set of classes that give tools for programmers to more easily create graphical user interfaces. It also includes an extensive hypertext documentation that includes methods, parameters, what the methods return, and examples of how to use the objects. Libraries and APIs are created for the programmers, but the end user benefits from quicker and cheaper development cycles. Libraries have been used by software engineers for decades, and are becoming increasingly sophisticated. Recent work has explored how libraries can be more useful to programmers. We support our library API understanding through the use of direct metaphors to the scientific meanings, such as an Earcon class representing an earcon concept. We also background almost all of the differences between our code and typical Java widgets. Therefore, the programmer can learn the metaphors only if he has to dig deeper than basic widget use.

Finally, we explored existing audio widget libraries. These were analyzed in terms of the first two areas: the widgets supported and the API usability. Many of these libraries offer specific tools that do an excellent job at completing their task. However, we feel that the APWidgets library offers a solution which is more compatible, flexible, and usable than many of the alternatives.

It is important to reiterate that several large scale audio systems, such as MERCATOR [2] and Pink [1], have been considered during the development of APWidgets. However, our discussion here is limited to using auditory widgets in a largely graphical programming world.

2.1. Audio Concepts and Widgets

The following discussion of auditory widgets is aimed at a basic understanding of the auditory interface concepts, and some thoughts about how to implement those ideas in widgets. While many audio concepts and widgets have been conceived, this part will only cover those which we explored with APWidgets.

Auditory icons are natural-sounding representations of objects. The graphical analog is an icon, which visually represents the object it emulates. For example, a video file might be represented by a movie reel as an icon and the clicking sound of the moving reel as an auditory icon. Gaver introduced auditory icons [14] and explored their use in the SonicFinder auditory interface [15]. From an implementation standpoint, the auditory icons could be prerecorded sounds played back on an event. They could also be dynamically generated based on particle modeling.

Earcons capture less about the object itself but more about its relation to other elements. For example, if a menu had a tone of a piano playing the A note, a submenu could have the A note

followed by a B note. Blattner et. al. introduced earcons [16]. Their work explored grouping earcons to give the user hierarchical information. In user testing, Brewster [11] determined that earcons could effectively convey information to the end user. Brewster also determined that earcons were more effective when the sound was a musical note instead of a pure tone. From a programming standpoint, the earcon possibilities include prerecorded sounds and dynamically generated MIDI.

Spearcons are a non-speech audio representation of a spoken phrase [12]. Generally, a text phrase is converted to speech via a text-to-speech (TTS) synthesis, and the TTS phrase is sped up dramatically, to the point where it may even be no longer recognized as speech at all. Alternatively, a recorded audio file containing the spoken phrase can be used as the base, and then sped up directly. Either way, the resulting spearcon is like a fingerprint of the original text phrase [12]. Spearcons have been shown to be useful as enhancements to menus, and as such can also be followed in a menu by the uncompressed TTS phrase if the user might need access to the full text message. Walker and colleagues [12, 17] showed that spearcons produced more accurate responses from participants compared to auditory icons or earcons and an equal accuracy when compared to TTS. Spearcons can be implemented dynamically with a Text-to-speech engine or use prerecorded speech sounds for the final product.

Soundscapes are auditory scenes. They can be natural, such as the sound in a park, or synthesized. The purpose of a soundscape can be aesthetic or informational. TAPESTREA [18] and SoundScape [7] are just two examples of tools designed specifically for building soundscapes. In general, soundscapes can be implemented through particle modeling, additive synthesis, physical modeling, generated MIDI, or using recorded sounds.

There are several design guidelines for building auditory widgets [19, 20]. Recently, Yalla and Walker's auditory menu review summarized the logical structure and activities afforded by menus [21]. These guidelines often provide a mix of software and audio design. APWidgets focuses on the software implementation. Therefore, the audio design for files which can represent auditory icons and earcons is still rudimentary in the current implementation. This follows the general Java implementation, where items such as icons are expected to be generated separately by an interface designer or graphical artist, and supplied to the program.

Table 1 briefly describes reasonable ways to implement each sound type. The first row of letters indicates approaches that could be the most flexible. The second row indicates approaches which could be used in some, but not all, situations.

Each implementation approach has its own tradeoffs. Recorded sounds can provide high-fidelity, controlled experiences generated by the programmer. The files require storage space. Recorded sounds are also limited to the sounds generated at the time of the software release. The other approaches all offer dynamic, run-time sound generation.

Physical modeling is an attempt to simulate the sound generation process of a natural environment. For example, maracas could be modeled in a CAD environment as an ellipsoid with a handle and round beads inside. By moving the maraca shell, the user moves the beads inside and a lifelike sound is generated [22]. It is more flexible than recorded sounds, but it is only as good as the simulation. A more detailed simulation requires additional processing, and the sound quality overhead must be balanced with performance considerations.

MIDI is a computer-readable musical notation. Instead of capturing a waveform, MIDI generates sound through commands such as instrument selection and note specification [23]. It was created when bandwidth and processor speed was low, and generates relatively compact notation. When it is time to play the sound, the MIDI is sent to the sound card, which then generates the music. Since each sound card and operating environment use different ways to generate the sound, there are audible differences between two different renderings of the same MIDI information.

Text-to-speech (TTS) is a special derivative of physical modeling. Given a textual input, a TTS engine will attempt to create and speak words. Often, TTS engines are capable of modifying the speech properties such as voice quality and speed. Human listeners can often determine that a TTS engine is synthesized, but understandability is good enough for TTS-dependent computer users to rely on the tools for information.

Sound Type				
Auditory Icons	Earcons	Spearcons	Spoken Text	Soundscapes
RP -	RM P	RT P	RPT -	RP -

Table 1. *Reasonable ways to implement sound types. R = recorded sound, P = physical modeling, M = MIDI generation, T = Text-to-speech generation*

2.2. Usable APIs and Design Considerations

There are several papers on API usability [24-26]. They suggest cognitive and heuristic approaches to measuring usability. APWidgets has components which follow new metaphor, Java concepts, or audio research metaphors. The new metaphors encapsulate the back end processing and playing of the sound events. These will be explained in more detail in Section 3.

The Java concepts are Swing components selected for enhancement. Each widget can be called and extended as before, with an "AP" before the widget name. This leads to functional programs with an API that may never need to be understood by the programmer. "API transparency" refers to this approach; the API is invisible to the programmer unless an uncommon change is required. If changes to the regular functionality are necessary, then the programmer must learn some of the new metaphors. By leveraging transparency, we effectively remove most learning difficulties for basic library use.

Audio research metaphors include TTS, earcons, auditory icons, spearcons, and soundscapes. These concepts are turned into coded representations in APWidgets; for example, an earcon concept is instantiated by constructing an Earcon object.

2.3. Audio Libraries

There are several existing audio libraries, but they fail to be used for general software production. Some approaches have been too specialized to a particular implementation. Others provide design patterns and guidelines but no actual implementations. APWidgets captures many auditory widget ideas in a completed library and makes their use transparent or metaphorical.

3. SYSTEM DEISGN

APWidgets was designed to provide a base for Java programmers to easily implement audio interface ideas into their programs. The event-driven approach features a sound manager that accepts incoming sound requests and decides whether to honor them or not. The sound manager then sends a threaded request to sound creation tools, including MIDI, TTS, and standard audio playback. A listener wiring tool connects each of the audio components to the sound manager behind-the-scenes.

The sound rendering is completed with third-party tools. Sound file playback is implemented via the Java API. MIDI generation uses Java Sound and JFugue [27]. Text-to-speech generation is achieved through Java Sound, JSAPI, and FreeTTS [28].

3.1. Development Goals

There were several goals in creating this software. First, it would be transparent. Other than including the library and using the widget, the programmer was shielded from the inner workings of the library. Even metaphors were avoided so that the programmer didn't have to learn anything extra. A typical widget is instantiated the same way as its extended class. For example, in the following code snippet, adding "AP" to the button constructor is the only real difference between a regular Java button and an APWidgets button:

```
JButton traditionalButton
   = new JButton("Visual Button");
APJButton audioEnhancedButton= new
   APJButton("AudioVisual Button");
```

Figure 1. *Widget name differences.*

Second, the library reflects many audio ideas. Concepts such as soundscapes, audio icons, earcons, and spearcons are each defined classes which can be used to enhance the application. As the science of auditory display progresses, this library can reflect the new changes.

Third, the library is easily extensible. If the audio libraries are insufficient, a programmer can change or extend the source code.

Finally, the library is robust. The architecture depends on tested third-party libraries for final rendering and the event model for delivering sound trigger messages. Java was selected since it has a strong cross-platform appeal.

3.2. Wiring for Sound

The new sound interpretation model differs from that of the graphical model in Java. With visuals, a component is given a Graphics object which can draw only in the bounds of the component space. In order to reduce interference between sounds, we built a system that operates on an event model. When a widget wants to make a sound, it sends a request in the form of a sound event. The sound event has sound settings that specify the type of sound (such as MIDI or spearcon) and parameters related to that type (such as words per minute). This sound event is sent via a sound listener to the sound manager. The sound manager ultimately determines what is played. The sound manager then sends specific playback information to MIDI, TTS, and waveform players. The sound manager is threaded in order to manage multiple requests. Figure 1

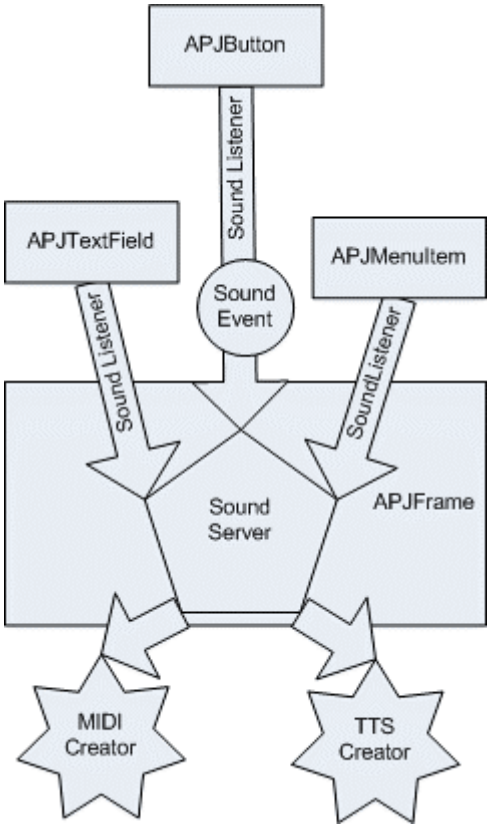

Figure 2. *The process of triggering a sound event.*

describes the process of triggering a sound event. APComponents inform their SoundListener with a SoundEvent that contains SoundSettings, or details of the event. The SoundServer, located in the APJFrame, receives these events. It then informs the appropriate sound generation tools what to do.

When a programmer is developing his software, he does not need to worry about the sound management. This entire process is going on behind the scenes in the library. When new components are added or changes are made to the user interface class structure, a sound wiring class searches for components that accept the sound listener and the wiring tool deploys a sound listener there.

The SoundServer class acts as the sound manager. It could easily be extended to accept new types of sounds as new ideas develop. Its playback logic affects the user experience much like a graphical user interface's canvas: the layout, display, and what is not displayed shape the aesthetic and usability qualities of the system.

3.3. Sound Types

There are three basic types of sound playback supported: waveform files, MIDI musical notation, and text-to-speech (TTS). A more complex inheritance hierarchy exists in the class structure, but this is basically how the notes will be played.

The Talker TTS class supports plain TTS and spearcons. Talker takes in a string and speech parameters and outputs a spoken rendition of that text. It is limited to one voice, provided by the FreeTTS library. Pitch range, volume, and speed can be modified. Due to the nature of TTS processing, the support libraries range from 10 to 100 Megabytes in size. In

order to facilitate flexible use of APWidgets, we decided to use the lightweight and open source FreeTTS [28].

The SoundFile class contains the general waveform loading and playback. Auditory Icons and Earcons are essentially a predefined group of waveforms from a particular class of sound. As mentioned earlier, Auditory Icons are context-aware sounds. Earcons are context-free sounding, but contain structural information such as the current location on a menu. From an implementation standpoint, the only difference is the class of sounds used.

Finally, APWidgets contains a few classes that manage MIDI generation. The JFugue MIDI music library [27] renders sound while the API provides tools for quickly generating a few musical notes.

In all cases, the focus was to make specialized tools that generate sound within a reasonable amount of time from when the activity went off. Time-to-play of more than a few hundred milliseconds was unacceptable, and the code was optimized until we reached that standard.

3.4. Sound Widgets

APWidgets depends on the certain auditory concepts. While the optimal strategy of implementing these ideas is disputable, our goal was to create a reasonable starting implementation and improve it over time. Earcons, auditory icons, and soundscapes depend on prerecorded audio files for sound playback. Spearcons and TTS depend on TTS playback. MIDI generation was used for some nominal sounds.

All of the sound widgets are able to take a sound listener and trigger a sound event when required. They inherit this ability from an abstract APComponent class. All APComponents accept wiring from the sound wire tool.

We note that our efforts in making sound-enhanced widgets did not focus on accessibility at this time. The current design was intended to bring basic and reasonable sound effects which have led or could lead to improved user performance and experience. Ongoing development includes accessibility features, among others.

We have extended several standard Java Swing widgets. The button, check box, scrollbar, slider, text component, and text field changes add MIDI sounds to various actions. The APJFrame acts as a container for the sound manager. The wiring tool searches for children and further ancestry of the APJFrame and adds listeners to the APComponent objects found. APJMenu and APJMenuItem are implemented with spearcons, but accept earcons and auditory icons as well. A new widget, APJGriddedPanel, gives map-like coordinate feedback to where the mouse is on the picture-panel.

4. USE EXAMPLE

This work emphasizes creating tools that enhance usability while minimizing the cognitive load on the programmer. The following Java code example displays both.

First, the APJFrame is initialized. This contains the sound manager class and wiring tools. As widgets are added to the frame, the frame checks (in the background) if they or their children are audio enhanced components. If so, the sound manager delivers a listener to the component so that it can inform the manager of any sound events. After all of the components are set up, a wiring call is made to double check listener connections.

The primary difference between this code and a typical Java Swing application without the sound are the wiring calls and the

```
public class APWidgetsDriver {
public static void main(String[] args) {

// initialize the frame and SoundServer.
APJFrame frame = new APJFrame();

// menuing
APJMenuBar bar = new APJMenuBar(frame);
frame.setJMenuBar(bar);
APJMenu file = new APJMenu("File");
file.add(new APJMenuItem("New"));
file.add(new APJMenuItem("Open"));
file.add(new APJMenuItem("Exit"));
APJMenu edit = new APJMenu("Edit");
edit.add(new APJMenuItem("Cut"));
edit.add(new APJMenuItem("Copy"));
edit.add(new APJMenuItem("Paste"));
bar.add(file);
bar.add(edit);

// some extra components
APJButton button
    = new APJButton("Just a button");
APJTextField text = new APJTextField();
JPanel southPanel = new JPanel();
southPanel.setLayout(new BorderLayout());
southPanel.add(button, BorderLayout.EAST);
southPanel.add(text, BorderLayout.CENTER);
frame.add(southPanel, BorderLayout.SOUTH);

// search from the button to the frame.
button.findSoundListener();

// gridded panel display
APJGriddedPanel drawingPane
    = new APJGriddedPanel();
JLabel label = new JLabel();

// put in your image file here!
label.setIcon(
    new ImageIcon("images/space.jpg"));
drawingPane.add(label);
JScrollPane scrollPane
    = new JScrollPane(drawingPane);
scrollPane.setVerticalScrollBar(
    new APJScrollBar());
scrollPane.setHorizontalScrollBar(
    new APJScrollBar());

frame.add(scrollPane,
    BorderLayout.CENTER);

// wrap it all up,
// including a sound wiring check.
frame.wireSoundSystem();
frame.setSize(new Dimension(1024, 768));
frame.setDefaultCloseOperation(
    APJFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}}
```

Figure 3. An *example of APWidgets in a program.*

"AP" before the component class names. If no wiring checks are done, then the difference boils down to two letters for each class. There are no concepts or even metaphors that the programmer must learn in order to use the sound enhanced toolkit.

5. CONCLUSIONS

The APWidgets software library provides Java programmers a way to implement audio widget ideas without the cost of learning the academic and implementation concepts behind the widgets.

AudioPlusWidgets is currently being evaluated on two levels. First, end users are performing benchmark tasks, such as map-searching, with both audio-enhanced and visual-only interfaces. This is providing an evaluation of the *utility* of the enhancements. Second, programmers are coding applications using typical Java Swing widgets, and with the APWidgets enhancements. This is allowing us to evaluate the *usability* of APWidgets. Together, these kinds of evaluations will form the next phase of this project, and results will be fed back into the library development efforts.

This work on AudioPlusWidgets is part of a larger effort to understand why auditory enhancements are still typically left out of software, and what can be done about it. Future work will include more evaluations of APWidgets, the development of widgets that depend on audio-based user interaction, and a comprehensive overview of how sounds are used in software. It is our hope that these new tools will be adopted and implemented widely, and expanded into a broadly supported means of bringing the potential benefits of audio-enhanced interfaces to great numbers of software applications.

6. REFERENCES

[1] T. Dougherty, "What Does Pink Sound Like? Designing the Audio Interface for the TalOS," in *International Conference on Auditory Display*, Palo Alto, NM, USA, 1996.

[2] W. K. Edwards, E. D. Mynatt, and T. Rodriguez, "The Mercator project: a nonvisual interface to the X Window system," *X Resour.,* pp. 33-53, 1993.

[3] P. Roth, L. S. Petrucci, A. Assimacopoulos, T. Pun, "Audio-haptic Internet Browser and Associated Tools for Blind and Visually Impaired Computer Users," in *Workshop on friendly exchanging through the net*, 2000.

[4] T. Hermann, C. Niehus, and H. Ritter, "Interactive Visualization and Sonification for Monitoring Complex Processes," in *International Conference on Auditory Display*, Boston, MA, 2003, pp. 247-250.

[5] S. Carla and B. C. Alan, "Using sound to extract meaning from complex data," in *Proceedings of SPIE,* 1991, pp. 207-219.

[6] S. K. Lodha, J. Beahan, T. Heppe, A. Joseph, and B. Zane-Ulman, "MUSE: A Musical Data Sonification Toolkit," in *International Conference on Auditory Display*, Palo Alto, NM, USA, 1997.

[7] B. S. Mauney and B. N. Walker, "Creating Functional and Livable Soundscapes for Peripheral Monitoring of Dynamic Data," in *International Conference on Auditory Display*, Sydney, Australia, 2004.

[8] K. van den Doel and D. K. Pai, "JASS: A Java Audio Synthesis System for Programmers," in *International Conference on Auditory Display*, Espoo, Finland, 2001.

[9] B. L. Vercoe, "C-sound," Experimental Music Studio, Media Laboratory, Massachusetts Institute of Technology, Boston, MA 1985.

[10] Freedom Scientific, "JAWS for Windows Overview." http://www.freedomscientific.com/fs_products/software_jaws.asp

[11] S. A. Brewster, P. C. Wright, and A. D. N. Edwards, "An evaluation of earcons for use in auditory human-computer interfaces," in *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems* Amsterdam, The Netherlands: ACM, 1993.

[12] B. N. Walker, A. Nance, and J. Lindsay, "Spearcons: Speech-based Earcons Improve Navigation Performance in Auditory Menus," in *International Conference on Auditory Display*, London, UK, 2006, pp. 63-68.

[13] J. Lumsden and S. Brewster, "A Survey of Audio-Related Knowledge Amongst Software Engineers Developing Human-Computer Interfaces," Glasgow University, Glasgow, England 2001.

[14] W. W. Gaver, "Auditory Icons: Using Sound in Computer Interfaces," *Human-Computer Interaction,* vol. 2, pp. 167-177, 1986.

[15] W. W. Gaver, "The SonicFinder: An Interface That Uses Auditory Icons," *Human-Computer Interaction,* vol. 4, pp. 67-94, 1989.

[16] M. M. Blattner, D. A. Sumikawa, and R. M. Greenberg, "Earcons and Icons: Their Structure and Common Design Principles," *Human-Computer Interaction,* vol. 4, pp. 11-44, 1989.

[17] D. K. Palladino and B. N. Walker, "Learning Rates for Auditory Menus Enhanced with Spearcons Versus Earcons," in *International Conference on Auditory Display*, Montreal, Canada, 2007, pp. 274-279.

[18] M. Ananya, R. C. Perry, and W. Ge, "TAPESTREA: sound scene modeling by example," in *ACM SIGGRAPH 2006 Sketches* Boston, Massachusetts: ACM, 2006.

[19] J. Lumsden and S. Brewster, "Guidelines for Using the Toolkit of Sonically-Enhanced Widgets," Glasgow University, Glasgow, Scotland 2001.

[20] J. Lumsden and S. Brewster, "Guidelines for Audio-Enhancement of Graphical User Interface Widgets," in *Proceedings of HCI*, London, England, 2002.

[21] P. Yalla, "Advanced Auditory Menus," Georgia Institute of Technology Tech Report, Atlanta October 2007.

[22] P. R. Cook, "Physically Informed Sonic Modeling (PhISM): Synthesis of Percussive Sounds," *Computer Music Journal,* vol. 21, 1997.

[23] MIDI Manufacturers Association Incorporated, "The Technology of MIDI: MIDI Files." http://www.midi.org/about-midi/abtmidi2.shtml

[24] S. Clarke, "Measuring API usability," in *Dr. Dobbs Journal*, May 2004.

[25] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building More Usable APIs," *IEEE Software,* vol. 15, pp. 78-86, May 1998 1998.

[26] C. Bore and S. Bore, "Profiling Software API Usability for Consumer Electronics," in *International Conference on Consumer Electronics*, 2005, pp. 155-156.

[27] D. Koelle, "JFugue," http://www.jfugue.org.

[28] S. M. L. Speech Integration Group, "FreeTTS,". http://freetts.sourceforge.net/docs/index.php.