

ENO: Synthesizing Structured Sound Spaces

Michel Beaudouin-Lafon
L.R.I. - CNRS URA 410
Bât. 490 - Université de Paris-Sud
91 405 Orsay Cedex
FRANCE
e-mail: mbl@lri.fr

William W. Gaver
Rank Xerox EuroPARC
61, Regent Street
Cambridge CB2 1AB
ENGLAND
e-mail: gaver@europarc.xerox.com

ABSTRACT

ENO is an audio server designed to make it easy for applications in the Unix environment to incorporate nonspeech audio cues. At the physical level, ENO manages a shared resource, namely the audio hardware. At the logical level, it manages a sound space that is shared by various client applications. Instead of dealing with sound in terms of its physical description (i.e., sampled sounds), ENO allows sounds to be represented and controlled in terms of higher-level descriptions of sources, interactions, attributes, and sound space. Using this structure, ENO can facilitate the creation of consistent, rich systems of audio cues. In this paper, we discuss the justification, design, and implementation of ENO.

Keywords: Auditory interfaces, multimodal interfaces, client-server architecture, sound

INTRODUCTION

In this paper, we describe *ENO*, an audio server designed to make it easy for applications in the Unix environment to incorporate nonspeech audio cues. Such cues can be useful in providing feedback about user-initiated events, notifications of events caused by the system or other users, and peripheral awareness of a distributed community of users. A number of different strategies have developed for designing audio cues, and basic research has started to describe the structure of everyday sound-producing events. However, though a number of systems have been developed which demonstrate the utility of auditory interfaces, these have been developed on an ad hoc basis and are largely restricted to using sampled sounds. ENO provides higher-level control over the development of auditory interfaces and allows the use of synthesized everyday sounds.

To provide background for ENO, we begin by outlining the use of sounds in interfaces, focussing especially on nonspeech audio cues. We sketch some of the functions that such cues may provide, discuss strategies for developing auditory cues, and describe how people hear sounds in the everyday world. From this foundation, then, we build our description of ENO.

SOUND AND COMPUTERS

Three basic categories of sound have been used in computers. The first is speech: computers have been used as a medium for speech communication, and speech has been used both for input to and output from computers. The second is music: computers can be used as a recording medium, as controllers for external music-making devices, or as tools for creating music via synthesis (e.g., Roads & Strawn, 1985). The final category is nonspeech audio cues: simple beeps and buzzes have been used as alerts for decades, while attention has recently been urged to more sophisticated methods of conveying information with sound (e.g., Buxton, Gaver, and Bly, 1991).

Applications using speech, music, or audio cues can be distinguished according to whether sound is represented *physically*, in terms of samples describing a waveform over time, or *semantically*, in terms of the sounds' higher-level structure. For instance, teleconferencing and voice mail systems need not (and often cannot) be concerned with the content of a given utterance (see e.g., Hindus & Schmandt, 1992). Similarly, applications that simply store and retrieve music need not be concerned with whether it is Mozart or Nirvana. In both cases, low-level physical representations suffice. But higher-level semantic representations are necessary for many other applications. For instance, both voice recognition and speech synthesis systems rely on phonemic and semantic representations of language. Music synthesis and sequencing programs rely on higher-level representations of musical structure as embodied, for instance, by the MIDI standard. In general, higher-level representations allow more powerful and relevant applications to be developed than low-level physical ones.

Most applications which use nonspeech audio cues have not explicitly represented their meaning, but instead simply used sampled sounds. In building ENO, we are concerned with bringing the power of higher-level semantic representations to the development of auditory interfaces. This requires the development of a structure for describing and manipulating sounds in terms of their content. Previous work, including both basic research and applications, provides useful hints about what this

structure should be. Using this structure, ENO can facilitate the creation of consistent, rich systems of audio cues.

Functions for Audio Cues

Over the last several years, a number of applications have been developed which explore the potential of nonspeech audio cues to provide useful information. Three basic sorts of function have emerged:

- *feedback*: what you're doing
- *notification*: what the system is doing
- *awareness*: what other people are doing

For instance, the SonicFinder (Gaver, 1989) extended the Macintosh desktop metaphor with sounds that provided feedback about user actions. Selecting a file, for example, produced a sound like tapping on an object. This provided feedback that the system had registered the event, and in addition notified the user about non-visible properties of the file such as its size and type.

A later system, the ARKola bottling plant (Gaver, Smith & O'Shea, 1991), used sounds for feedback, but also incorporated many sounds used as notifications about events in the system. For instance, as virtual cola was cooked, bottled, and capped in the system, each virtual machine made its own processing sound that notified the user about its activity and rate. This allowed users to monitor processes that were not visible on their screens.

Finally, the EAR system (Gaver, 1991) works in conjunction with an "event server" called Khronika (Lövstrand, 1991) to notify members of EuroPARC about events in the laboratory. Because these are often communal events, and because the sounds are widely accessed by members of the lab, the sounds help colleagues maintain awareness of one another's activities. For example, sounds are used to indicate that someone has accessed the signal from a local video camera, that a meeting is about to begin, that it has started to rain, even that friends want to go to the pub.

Note that the functions of a given sound may be different for different users. For instance, in a collaborative system a sound providing feedback to one user may serve to notify another about changes in the system or to increase their awareness of other users. This is the case with GroupDesign (Beaudouin-Lafon & Karsenty, 1992), a shared editor which uses the notion of an *audio echo* both to provide feedback and to support other users' awareness of colleague's actions. Nonspeech audio cues may thus play a number of different roles in computer systems.

Methods for Using Audio

In order to develop systems of nonspeech audio cues, a strategy must be found that constrains the mapping of sounds to information, and guides the choice of particular sorts of sounds. Several such strategies have been suggested over the last several years. *Auditory icons* (Gaver, 1986, 1993) rely on an iconic mapping between computer events (e.g., selecting a file icon) and everyday sound-producing events (e.g., tapping a solid object). *Earcons* (Blattner et al., 1989) rely on a conventional mapping between computer events and simple motives (short tunes like those used in *Peter and the Wolf*). Finally, *genre sounds* (Cohen, 1993) rely on metaphorical mappings between computer events and associated events drawn from movies, TV shows, or conventional situations.

Each of these strategies has characteristic strengths and weaknesses. Auditory icons rely on users' knowledge of everyday sounds and so can be designed to be obvious and easy to learn, but the appropriate sounds may sometimes be difficult to find. Earcons can be systematically generated and designed to be maximally discriminable (Brewster et al., 1993), but they may be difficult to learn and remember and may more difficult to fit with the graphical components of the interface. Finally, genre sounds strike a compromise between auditory icons and earcons: they may share the clarity of auditory icons and the flexibility of earcons, but may equally share the problems of finding appropriate mappings experienced with auditory icons and the difficulties of learning and awkwardness of earcons. In general, the appropriate strategy would seem to depend on the particular application; in some cases, the strategies might usefully complement each other.

Structuring Sound

For any of these strategies, the creation of auditory interfaces is facilitated by an understanding of how sounds are or may be structured. In this way, systems may be built which support their creation in terms of higher-level representations. Here we briefly describe the structure of sound in terms of events and sound space.

Events

Basic research on everyday listening has allowed the development of a framework for describing everyday sounds that can be applied to auditory icons (Gaver, 1993). According to this approach, sounds are heard in terms of the *sound-producing events* that cause them. These events, in turn, are comprised of *interactions of materials*. Both interactions and materials have characteristic attributes (e.g. interactions might be characterized by their force or duration; materials by their size or hardness). Events can further be organized according to whether the materials involved are

solids, liquids, or gasses, and whether the interactions are simple *basic-level* events or more complex *temporally-patterned*, *compound*, or *hybrid* events (for instance, a single impact is a basic level event, while bouncing is the result of temporally-patterned impacts). More complex events are assumed to inherit the attributes of their simpler components and to have new idiosyncratic attributes of their own. Using this approach, a framework can be developed that describes a large space of everyday sounds succinctly.

Although this framework developed out of basic investigations into everyday listening, the overall categorization scheme works well for earcons and genre sounds as well. In each case, a basic "source" can be identified – for everyday sounds, the material; for earcons, the base motive; for genre sounds, the sound effect. This source makes sounds when it is involved in some event – for everyday sounds this depends on the interaction; earcons and genre sounds depend simply on the source being triggered. Finally, several attributes of the event may be varied to produce a class of sounds around a central source – for everyday sounds, variations in the material or interactions will produce different sorts of sounds; for earcons new messages can be produced by introducing systematic variations to existing motives.

Sound Space

We don't just hear isolated events, but a *sound space* of events occurring at locations in one or several environments. Events may be joined or separated by the proximity of their sources or the similarities of their environments. This property of sound spaces has useful implications for applications (see, e.g., Cohen and Ludwig, 1991). For instance, notifications about similar interface events might be heard from the same virtual location, while other events could be heard from elsewhere. Similarly, urgent messages could be made to sound close, while more general information could be designed to sound further away. Systems have been developed which allow the creation of realistic, 3D sound spaces (Wenzell et al., 1991; Burgess, 1992), but the computational demands are very high. In the work reported here, we develop a much simpler, 2-dimensional sound space which nonetheless seems to provide many of the benefits of more sophisticated systems.

In general, understanding sound in terms of sources, interactions, attributes, and sound space is a promising foundation for organizing and controlling auditory interfaces. Sounds may be generated and varied in flexible ways, and a more richly structured sound environment can be supported than systems that merely allow the playback of sampled sounds. In the remainder of this paper, we describe how this structure is used in ENO.

We start by describing ENO itself, then describe how client applications interact with it, discuss several challenges for its implementation, and end with brief descriptions of three auditory interfaces that use this system.

THE ENO AUDIO SYSTEM

We have implemented ENO under Unix as a server and a library used by clients to communicate with the server. The architecture is very similar to that of the X Window System (Schiefler and Gettys, 1986). ENO makes it easy for client applications to use audio by providing them with high-level control over the sound characteristics and rendering process. At the physical level, ENO manages a shared resource, namely the audio hardware. At the logical level, it manages the sound space, shared by the various client applications.

Sources and the Source Tree

ENO's architecture reflects the framework for describing sound introduced earlier. The central concept in ENO is that of a *sound source* (or *source* for short). Sources represent possible causes for sound rather than sound itself. They produce sound as the result of *interactions*. For example, a source that represents a bell will produce one sort of sound when an impact interaction is applied to it, and a different sort of sound when a scraping interaction is applied. Sound sources are characterized by a type and a set of attributes. Interactions are characterized by a type and a set of parameters.

The sound sources are arranged in a tree (see figure 1). Sources are created by client requests, except for the root of the source tree, which is created by the server. The purpose of the sound tree is two-fold. First it provides an inheritance mechanism for attribute values. Second it defines a control structure for generating sounds: leaves of the tree, called *primitive sources*, produce sound that may be controlled by upper-level nodes of the tree, called *high-level sources*.

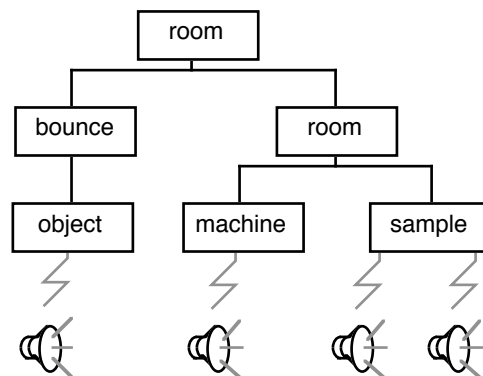


Figure 1: A sound tree.

Attribute inheritance

Attributes are inherited along the branches of the sound tree. If a sound source does not define an attribute, this attribute is looked up in its parent. Some attributes are relative, which means that the value used by the sound source for that attribute is the combination of the values of the attribute along the branch between the sound source and the root of the sound tree.

This mechanism provides simple sharing of values and allows for local control over a set of sources. For example, all sources have a relative attribute, the gain, which represents an attenuation (if negative) or enhancement (if positive), in dB. The actual gain applied to a sound is the algebraic sum of the gains of the sources between the root and the source playing the sound. This means that the volume of a collection of sources can be controlled by changing the gain of their common ancestor.

Primitive sources

The leaves of the sound tree are the only sources that directly produce audio output. So far we have implemented three types of primitive sources: sampled sounds, objects, and machines. A sampled sound source plays a sound stored in a file referenced by a name looked up in a set of server-dependent directories. Object and machine sources use synthesis algorithms to produce auditory icons (Gaver, 1993). With these algorithms, attributes of the source and parameters of the interaction can be changed while the sound is playing, providing client applications with high-level real-time control over the sound. This contrasts with sampled sounds which are, by definition, predefined and unchangeable.

Object sources are defined by attributes describing physical parameters such as the shape, size and material of the object. Object sources produce sounds with two types of interaction: impacts and scrapes. The parameters of an impact are the hardness and force of the virtual mallet that hits the object, while the parameters of a scrape are the roughness of the scraper and the speed of the scraping.

Machine sources produce continuous sounds evoking engines. Their attributes include the size and timbre of the machine. The parameters of the interaction with the machine source are the speed of the engine and the amount of work it delivers.

High-Level Sources

The nodes of the source tree provide the basic structuring mechanism for the sound space. They are used to group primitive sources both temporally and spatially. This allows the creation of complex sound-producing events from basic

level components, and the definition and control of a structured space of sound events.

Complex Sound-Producing Events

High-level source nodes are used to create complex sound-producing events from their basic level components (see Gaver, 1993). Three types are currently available: sequential, parallel and bounce. Sequential and parallel sources play their subsources in sequence or in parallel, respectively. A bounce source "bounces" its (unique) subsource; this is achieved by playing the subsource at logarithmically decreasing time intervals with logarithmically decreasing gain. An attribute of the bounce source defines the initial height, used by the server to compute the time and gain series. Using a parallel source to trigger several bounce sources can be used to create still more complex breaking sounds (Gaver, 1993).

Rooms in Sound Space

Another high-level source is the room, which provides a way of grouping sounds according to their location.

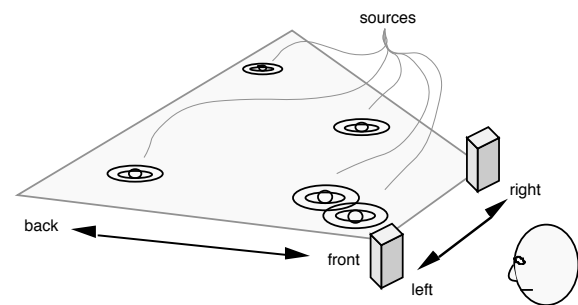


Figure 2: the sound space

The location is a coordinate in a horizontal plane (see figure 2). The coordinate on the left-to-right axis defines the position of the sound in the stereo field. The coordinate on the front to back axis defines the "depth" of the sound, or its distance from the foreground. A far source sounds muffled, while a close source sounds normal. The location in this 2-D space is not meant to provide the user with an accurate localization of the sound. In fact, since output from ENO is likely to be played through loudspeakers (we don't want to force users to wear headphones), this would be impossible. Instead, the purpose of the 2-D location is to make it easier for the user to distinguish and identify several sources playing at the same time.

Thanks to the tree structure, rooms can be nested. The location attribute is relative, so a set of sounds can be moved by changing the location of their common room ancestor. The root of the source tree is a room source.

PROGRAMMING ENO

Client applications access the server over a network connection. Access to the server is controlled by a list of hosts from which connections are accepted. The API used by client applications to communicate with the server is Enolib, a library written in C. We have also developed an extension to the Tk/Tcl toolkit (Ousterhout, 1990, 1991).

Every sound source is identified by a unique number (ID) and belongs to the client that created it. When a client disconnects from the server, its sources and their subtrees are destroyed. There is no access control mechanism: if a client knows the ID of a sound source, it can manipulate it.

With Enolib, clients applications can:

- create and destroy sound sources;
- create interactions with sources;
- query and change the values of sources attributes and interaction parameters;
- ask the server to be notified of certain events.

Sources and Interactions

To create a source, a client sends a request with its type and the ID of its parent in the source tree. The server creates the source and returns its ID. To create an interaction, a client sends a request with the ID of the source and the type and parameters of the interaction. For example, to create a scraping sound, a client specifies the object source ID, the scrape interaction type, and the speed and roughness of the scraping. The sound resulting from an interaction has a unique ID, returned by the server.

The distinction between source attributes and interaction parameters is important because a single source can be played several times, therefore producing several sounds that can play simultaneously. While source attributes are shared by all the sounds being played by that source, interaction parameters only affect one sound.

Attribute values of the sources can be changed at any time. If the modified attributes affect the sounds being played, the audio output reflects this in real-time. For instance, it is possible to "move" a source while it is playing by modifying its location. Similarly, one can control the overall volume of the output by changing the gain of the root source. Interaction parameters can be modified in a similar way, with real-time effect on the audio output. For example, the speed of scraping can be changed while the sound is playing.

Notification

The server can notify client applications by means of events. Clients can ask the server to be notified

when a source is modified, when a source is being played, and when a sound ends playing. They can also be notified of the creation and destruction of sources. These facilities serve different purposes. Notifications of sound start/stop are typically used by clients for synchronization purposes. Other notifications can be used by an audio-manager that would play the same role as the window-manager in a window system. For example, the audio manager could display each source in a window and provide a graphical interface for the user to control the location and gain of each source.

Examples

Figure 3 is a C program written with Enolib. It creates an object source, plays an impact sound on it and waits until the sound is finished playing.

The second example is written with Enotk, the Tk/Tcl (Ousterhout, 1990, 1991) extension that we have developed. Tcl is an extensible interpreted language; Tk is a graphical toolkit written as an extension to Tcl. A Tcl program consists of a set of lines, each of which consists of a command name and a set of command arguments. Commands between brackets are evaluated before the result is passed as an argument to the enclosing command. The programming style of Enotk is similar to that of Tk: sources are referred to by a path name: `.foo.bar` is a subsource of `.foo`, which is itself a subsource of `.`, the root source. There is one command for each source type, e.g.:

```
machinesource pathname -attr value ...
creates a machine source. Once a source is created, its name can be used as a command name:
```

```
# create a machine source
machinesource .printer -size 100
# play it
.printer play -speed 1000
# destroy it after 5 seconds
after 5000 {.printer destroy}
```

(`after` is a Tcl command that executes its second argument after a delay in milliseconds specified by its first argument)

In order to be able to refer to the sound once it has been created, the `play` command returns an identifier which can be used with the `sound` command to manipulate the sound:

```
# create a machine source
machinesource .printer -size 100
# play it and store sound ID in snd
set snd [.printer play -speed 1000]
# speed up the sound after 5 seconds
after 5000 {sound $snd -speed 2000}
# stop it after ten seconds
after 10000 {sound $snd stop}
```

(\$`snd` is the value of variable `snd`)

With Enotk, the example in figure 3 takes only two lines:

```
objectsource .obj -size 9 -material 5
.obj impact -hardness 50 -stop {exit}
```

```

#include <AF/AFlib.h>
main (int argc, char** argv) {
    AFAudioConn* aud = NULL;
    ASource      src;
    ASound       snd;
    AFEvent      ev;
    /* connect to server */
    if ((aud = AFOpenAudioConn ("")) == NULL) {
        fprintf (stderr, "%s: can't open connection.\n", argv [0]);
        exit (1);
    }
    /* create a source: a small object made of bright material (e.g. glass) */
    src = AFVaCreateSource (aud, AObjectSource, AFDefaultRootSource (aud),
                          AObjectSize, 9, AObjectMaterial, 5, 0);
    /* create an interaction: impact with medium hardness */
    snd = AFVaCreateSound (aud, AImpactSound, src, AImpactHardness, 50, 0);
    /* play sound, asking for notification at end of sound */
    AFPlaySound (aud, snd, APlayTimeRel, 200, APlayNotifyStop);
    /* wait until sound is finished playing */
    AFFlush (aud);
    while (AFNextEvent (aud, &ev))
        if (ev.type == ASoundStopEvent && AGetSound (&ev) == snd) return;
}

```

Figure 3: sample code using EnoLib

IMPLEMENTATION

The implementation of the ENO server is based on the AudioFile System (Levergood et al., 1993). We chose this approach to minimize the implementation cost and yet get a device-independent, network-transparent server. The server of the AudioFile System uses a protocol similar to that of the X Window System. In fact it uses a significant amount of code from the X11 server and libraries.

The main modifications that we made to the AudioFile server and library in order to implement ENO are the following:

- definition of server resources to represent sources and sounds. This ensures that source and sound IDs are unique and that they are cleaned up when a client exits;
- management of the source tree. The root source is created when the server starts and its ID is returned to each client during the initialization phase of the protocol;
- addition of the synthesis algorithms for the machine and object sounds;
- management of new events for the notification of clients by the server;
- addition of new requests to the protocol with the corresponding functions in the library.

No device-dependent code was changed, so a port of AudioFile to another platform would work for ENO unmodified. The rest of the modifications fall into two categories: local modifications spread out over the existing code (< 1000 lines of C code) and a module of new code which could be taken out to

build an entirely new server (on the order of 5000 lines of C++ code).

Role of the server

We chose to implement the structured sound model described in the first part of this article completely inside the ENO server. This differs radically from most other systems (Neville-Neil, 1993), which consider sound samples as the primary objects. With ENO, client applications never exchange samples with the server. Instead they create sources, edit their attributes, and create interactions with sources. Hence, the rendering process is the job of the server, not the client. This has several advantages:

- complex digital signal processing algorithms are implemented once, in the server, rather than in each application.
- the server can take advantage of specialized hardware, making client applications device-independent. In particular, issues such as sampling frequency and sample formats are completely invisible to the clients.
- the client-server protocol is lightweight, since the requests need not carry any sound data. This improves network-transparency since a protocol that requires less bandwidth is likely to be usable over a wider range of networks.
- time management is mostly the server's job. Most existing audio servers manage audio streams which require client applications to be (almost) real-time processes that deliver chunks of samples in a timely manner.

Implementation issues

The main difficulty in the implementation was to ensure real-time generation and control of sound.

This raised three issues: real-time operation, CPU overrun, and time-management.

Real-time operation

Unix is not a real-time operating system, and the network does not guarantee bounded-time delivery. Therefore, strictly speaking, there is no way to build a real-time system. The typical approaches to working around this problem are to increase the server's process priority (some flavors of Unix even provide non-decreasing priorities and real-time priorities) and to insert a constant lag between the time at which samples are computed by the server and the time at which they are played. However these are fixes, not solutions, and they often produce unacceptable performance – for instance, teleconferencing systems that use this strategy have audible delays comparable to those of satellite-relayed telephone conversations. In the long run, the only possible solution is for the operating system and the network to provide the appropriate functionality.

CPU overrun

The server has to synthesize and mix several sounds faster than the output sampling rate, or there is a risk of overrun which results in audible clicks. Most audio servers (including AudioFile) don't handle this issue and defer it to clients. This is unacceptable because clients do not have control over, and cannot even know, the server's processing load. We believe the server can better handle this issue, using at least three techniques:

- *caching*: since the same sources are likely to be played several times with the same interaction, caching the samples resulting from a computation saves time the next time the same sound is used. Caching can occur at several levels in the server. It is more appropriate to cache sounds before they are spatialized since the spatialization process is relatively cheap and the hit rate in the cache is much higher than if the spatialized sounds were cached instead.
- *adaptive synthesis*: the complexity of the synthesis algorithms can be evaluated precisely so that the computing time for synthesizing a given sound can be anticipated. This time complexity depends directly on the sample rate (usually linearly, sometimes more). The time complexity can therefore be reduced by synthesizing under-sampled sounds. The lower-quality sound that results from this technique is preferable to the audible clicks that result from CPU overrun. In order to minimize the degradation of perceived quality, the level of subsampling can depend on the localization of the sound: under-sampling distant sounds is less audible than under-sampling close sounds.
- *specialized hardware*: some hardware platforms already have specialized hardware for digital sig-

nal processing (dedicated DSP chip or a second RISC chip). We believe that this is likely to become more prevalent, just as dedicated graphics hardware has become common in most modern workstations. The server can take advantage of such hardware to increase the number and quality of sounds it can produce simultaneously.

Clearly none of these techniques can be implemented efficiently by clients. Also, it would be incompatible with the goal of simplifying clients' code. The present implementation of ENO uses a primitive caching technique for sampled sound sources: the samples are kept in memory as long as the source is active. There is also a primitive form of adaptive synthesis: distant sounds are synthesized at half the nominal sampling rate. Both techniques have proven effective in supporting the strategies outlined here.

Time-management

The AudioFile server requires the client to manage time explicitly: each request to play a batch of samples contains the time at which the batch must be played. After each such request, the server returns to the client the time that it received the request. If the request is received after the time to play, a number of samples of the request's batch are discarded. This mechanism is meant to facilitate the management of sound streams by clients. In our experience, it makes such programming overly complex and has one major caveat: one cannot tell the server to play a sound as soon as it can.

In ENO, time is represented by a mode (absolute or relative) and a time value. Absolute mode corresponds to the behaviour described above; Relative mode tells the server to interpret the time value as relative to the time at which it *receives* the request. Hence, a play request with a relative time of 0 tells the server to play the sound as soon as it can, i.e. the time at which it receives a request plus a constant delay to account for the processing of the request and the absence of real-time support. A delay of 100 to 200 ms has proven viable on a Sparc2 workstation; a delay of 50 ms can be handled reliably on an SGI Indy or HP 9000/715.

Synthesis Algorithms

The synthesis algorithms for the object and machine sources are adapted from (Gaver, 1993). Impact and scrape sounds are obtained by filtering an input wave by a bank of infinite impulse response (IIR) filters, while machine sounds are generated using FM synthesis. In both cases, the parameters of the synthesis algorithm (e.g. frequencies, bandwidth) are computed from the source attributes and interaction parameters. The reader is referred to Gaver (1993) for more details

and to Moore(1990) for an introduction to digital audio signal processing techniques.

The algorithm used by room sources to spatialize the sound in the 2-D space defined earlier is quite simple. Horizontal location is achieved by panning (i.e. using amplitude difference between the left and right channels) while depth is achieved by low-pass filtering (i.e. attenuating high-frequency components). This method has the advantage of being computationally efficient and gives a sense of volume. Although it would be desirable to use more sophisticated methods, 3-D sound techniques (Wenzell et al., 1991; Burgess, 1992) cannot be used because they work only with headphones and we would like the system to be effective using loudspeakers as well.

SAMPLE APPLICATIONS

We have developed several applications with ENO to evaluate both the potential of our model for structuring sound in standard applications and the ease of programming using ENO.

Audio Draw

The first application is a very simple drawing tool written with EnoTk. It uses object sources for the different tools and objects in the drawing. Standard interactions such as selection, drag and resize produce audible feedback, controlled in real-time. For example the speed of the scraping sound used when dragging objects tracks the mouse's speed. A shared version of Adraw uses the same cues for notification and awareness purposes.

Audio Make

The second application uses machine sounds to monitor the execution of the Unix *make* utility. With a simple modification of the makefile¹, each command that *make* runs plays a different machine sound for the duration of that command. The source and interaction parameters depend on the command's name and arguments, and evolve over time to produce the audio equivalent to a progress bar. This has proven effective in monitoring a make session in the background, especially since after a few runs of the same session, the user becomes accustomed to its sound and can monitor its progress.

Audio twm

The third application is an extension to the *twm* window manager for the X Window System. This extension uses real-time control over source arguments and interaction parameters to provide dynamic feedback and support navigation:

- impact and scrape sounds are used for feedback about the user's actions on the windows, icons and menus;
- impact sounds are played by the source that represents a window whenever the mouse cursor enters that window. This is helpful as a navigational clue to track the mouse's focus;
- a room source corresponds to each window whose location in the audio 2-D space is kept consistent with its location on the screen (left/right, top/bottom). With a simple inter-client protocol, X clients can create their sources under the room associated to their window, which results in transparent and consistent audio-visual display.

For both Audio Draw and Audio twm, the code that manages the audio portion of the application represents less than 5% of the total code. Audio Make is a 500 lines-long C program.

CONCLUSION AND FUTURE WORK

ENO provides a simple, powerful mechanism for creating auditory interfaces. There are a number of possibilities for the extension of this system. For instance, the synthesis algorithms for generating everyday sounds may be extended to incorporate new physical models, and to allow new virtual interactions to be used with existing virtual objects. In this way, the repertoire of everyday sound-producing events may be extended, and thus the ability to create rich and meaningful auditory icons.

The system is not limited to the use of nonspeech ambient sounds, however. The basic architecture of ENO is well-suited to the addition of a number of new sound sources to the system, including speech synthesizers, MIDI output for computer music, and systems of hierarchically varied earcons. In the end, ENO is not limited to being a system for creating auditory icons, or even for creating nonspeech audio cues. Instead it presents a new perspective on the much wider issue of supporting the use of sound – speech or nonspeech – in computer systems.

REFERENCES

- Beaudouin-Lafon, M. and Karsenty, A. (1992), Transparency and awareness in a real-time groupware system. Proc. ACM Symposium on User Interface Software and Technology (UIST'92, Monterey, November 1992), ACM Press, New York.
- Blattner, M., Sumikawa, D., & Greenberg, R. (1989). Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*. 4 (1).

¹ A simple modification of *make* itself would be more transparent but we did not have access to the program's source code.

- Brewster, S. A., Wright, P. C., and Edwards, A. D. N. (1993). An evaluation of earcons for use in auditory human-computer interfaces. *Proceedings of INTERCHI'93 (Amsterdam, The Netherlands, 24 - 29 April, 1993)*. ACM, New York, 222 - 227.
- Burgess, D. (1992). Techniques for low cost spatial audio. *Proc. ACM Symposium on User Interface Software and Technology (UIST'92, Monterey, November 1992)*, ACM Press, New York.
- Buxton, W., Gaver, W. and Bly, S. (1991). The use of non-speech audio at the interface. *Tutorial Notes, CHI'91 (New Orleans, April 28 - May 2, 1991)*.
- Cohen, J. (1993). "Kirk here:" Using genre sounds to monitor background activity. *INTERCHI'93 Adjunct Proceedings (Amsterdam, April 24 - 29, 1993)*. pp 63 - 64.
- Cohen, M., and Ludwig, L. (1991). Multi-dimensional audio window management. *IJMSS*, 34, pp 319 - 336.
- Gaver, W. W. (1986). Auditory icons: Using sound in computer interfaces. *Human-Computer Interaction*, 2, pp 167 - 177.
- Gaver, W.W. (1991). Sound support for collaboration. *Proceedings of the Second European Conference on Computer-Supported Collaborative Work (Amsterdam, September 24 - 27, 1991)*. Kluwer, Dordrecht.
- Gaver, W.W. (1993), Synthesizing Auditory Icons. *Proc. ACM Conference on Human Factors in Computing Systems (INTERCHI'93, Amsterdam, April 1993)*, pp 228-235, ACM, New York.
- Gaver, W.W., Smith, R.B., and O'Shea, T. (1991) Effective sound in complex systems: The ARKola simulation, *Proc. of Human Factors in Computing Systems (New Orleans, LA, April 29 - May 2, 1991)*, ACM Press, New York.
- Hindus, D. and Schmandt, C. (1992), Ubiquitous audio: Capturing spontaneous collaboration, *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'92, Toronto, Canada, October 31 - November 4, 1992)*, ACM Press, New York.
- Levergood, T., Payne, A., Gettys, J., Treese, W., and Stewart, L. (1993). AudioFile: A network-transparent audio server for distributed audio applications. *Proc. Summer Usenix Conference*, Usenix Association.
- Lövstrand, L. (1991). Being selectively aware with the Khronika system. *Proceedings of the Second European Conference on Computer-Supported Collaborative Work (Amsterdam, September 24 - 27, 1991)*. Kluwer, Dordrecht.
- Moore, F.R. (1990). *Elements of Computer Music*. Prentice Hall.
- Neville-Neil, G. (1993). Current Efforts in Client/Server Audio. *The X Resource*, Issue 8.
- Ousterhout, J.K. (1990). Tcl: An embeddable command language. *Proc. Winter Usenix Conference*, pp 133-146.
- Ousterhout, J.K. (1991). An X11 toolkit based on the Tcl language. *Proc. Winter Usenix Conference*.
- Roads, C., and Strawn, J. (eds.) (1985). *Foundations of computer music*. Cambridge, Mass., The MIT Press.
- Scheifler, R.W. and Gettys, J. (1986). The X Window System. *ACM Trans. on Graphics*, 5(1), pp 79-109.
- Wenzel, E., Wightman, F., and Kistler, D. (1991). Localization with non-individualized virtual acoustic display cues. *Proceedings of CHI'91 (New Orleans, April 28 - May 2, 1991)*. ACM, New York.