SoundWorks: An Object-Oriented Distributed System for Digital Sound

Jonathan D. Reichbach and Richard A. Kemmerer
University of California at Santa Barbara

SoundWorks lets users interactively manipulate sound through a graphical interface. The system handles digitally sampled sounds as well as those generated by software and digital signal processing hardware.

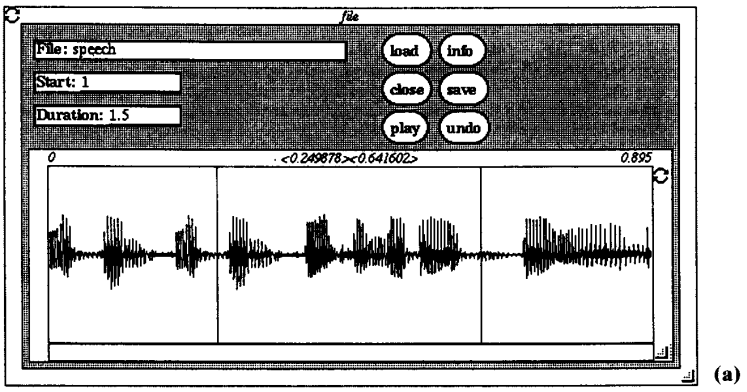
The field of computer-based music encompasses issues in music composition, synthesis, manipulation, and performance.[1-5] Here, we address the manipulation and synthesis of sounds. Our primary goal in this work was to provide standard sound manipulation (or editing) features like splicing, looping, and mixing. In so doing, we provided operations that could modify the amplitude, pitch, and duration of sounds. To aid in modifying sounds — and creating new ones — we predefined sounds representing basic sound-generating waveforms (for example, sine and triangle) for use with available operations.

Our second goal was to develop a server-based system that could be easily integrated with other applications and user interfaces, and that could be extended to support network-based access to sounds and devices. We addressed the large computational requirements of digital sound manipulation by integrating digital processing hardware into the system. This integration supports a more interactive environment for processing sounds and provides the possibility of real-time response.

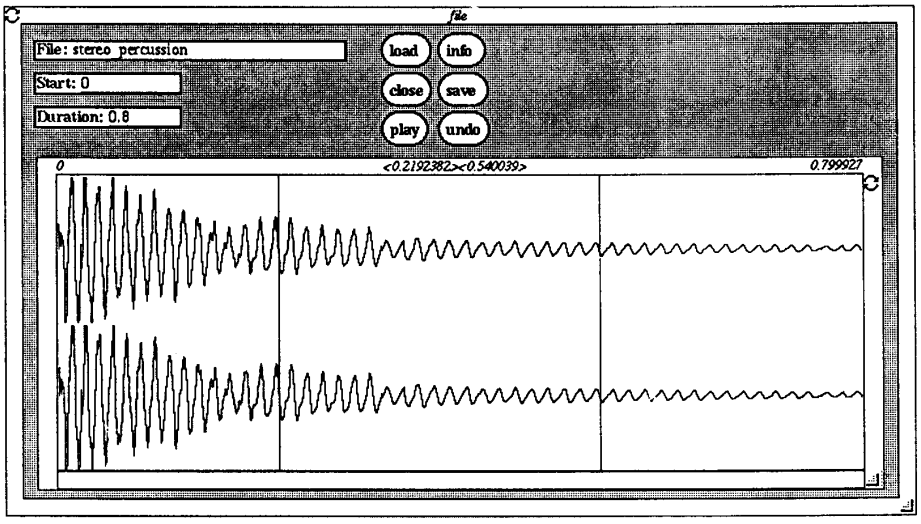
Because we wanted to have a server-based system with network-based access to sounds and digital processing hardware, we used Sun Microsystems' NEWS application programming environment[6] for developing the system. NEWS (which stands for network-extensible window system) provided the necessary primitive graphic items for a graphical window-based interface and let us use an object-oriented approach for development.

The resulting system, called SoundWorks, is an object-oriented distributed system for manipulating digital sound. It lets the user interactively manipulate these sounds with a graphical window-based interface and handles digital sampled

0018-9162/92/0300-0025$03.00 © 1992 IEEE



(a)



(b)

Figure 1. File sound windows for a mono sound (a) and a stereo sound (b).

sounds, sounds generated by software, or sounds generated by digital signal processing hardware. SoundWorks' distributed nature lets the sounds and digital hardware reside on a system other than the user's local system.

After a brief description of related research that provided the basis for SoundWorks, we introduce the SoundWorks system and present details about its design. Then we summarize the results of the project and discuss some areas for future research.

Related research

User interface design for sound processing has followed the trend for traditional applications. It progressed from punch cards and teletypewriter-based

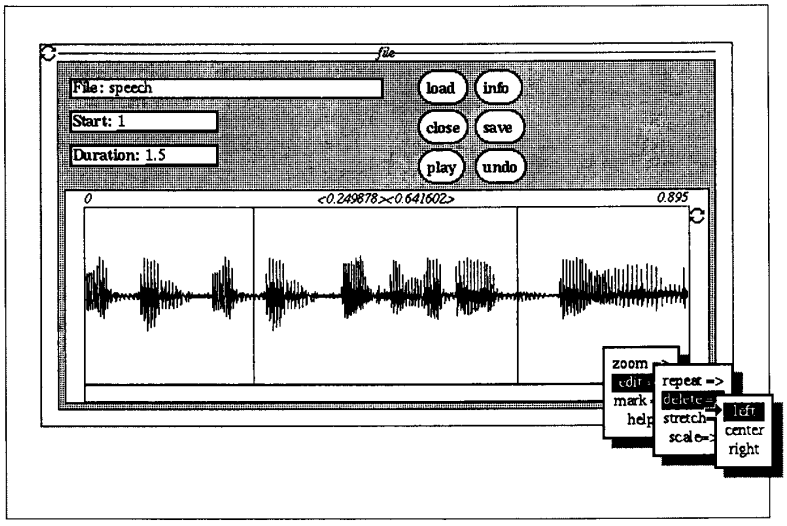


Figure 2. File sound window menu.

terminal interfaces to graphics-based window interfaces. These developments paralleled a transition from batch processing to more interactive environments. Also, with the advent of distributed systems, sound processing systems could be distributed to allow multiple users to share resources such as sound file systems and digital sound processing hardware.

User interfaces. Two issues important in developing a graphical user interface are consistency between the graphics model and the objects in the system, and compatibility with other graphics applications. A consistent system graphics model provides a close tie between objects and their corresponding graphic representations. For instance, a sound object can be displayed as a waveform, representing the change in amplitude over time.

This close tie should also extend to other objects in the system. For example, the graphic representation of a window should be consistent with that window's function, regardless of whether the window is open or closed. In SoundWorks we achieved this by using the same icon in different sizes to represent all windows that contain a sound. That is, a sound window can be either open, exposing the contents of the window, or closed, with an icon representing the contents of the window. Similarly, the help window icon is a picture of a text manual. The main goal is that the graphic objects, regardless of their state, represent the objects in the system.

Compatibility with other applications can be achieved with a graphics window-mouse package. This package provides a basic set of graphic objects (windows, menus, buttons, message boxes, and so on) that can be integrated with application-specific graphic objects. Developers can also provide a set of standard graphic objects representing sound-related objects. The HyperScore ToolKit,[7] MacMix,[2] and Sound Kit[3] are good examples of packages tailored toward sound applications. These systems provide a basic set of extensible graphic objects for displaying sounds, notes, and other application-specific information.

Following this approach, SoundWorks provides a set of graphic objects that can be used to represent and manipulate sounds. These graphic objects are easily extended and modified to provide a high-level interface for integrating audio and graphics.

Distributed sound systems. These recently developed systems provide better resource-sharing, reliability, and performance at lower cost. Some recent digital sound systems provide shared access to resources using local area networks. For instance, the Etherphone system[8] for network-based voice messages lets the user edit digital voice representations and play or record the result through special-purpose hardware at a local workstation. The Vox server,[9] in contrast, provides flexible configurations of digital hardware, such as speech synthesis, speech recognition, and video devices, which can be integrated with the user's local workstation. These devices can also be shared across the network.

The SoundWorks user interface, which resides on the user's local system, is distributed from the application code, which resides on the remote system where the sound files and hardware are located. The local graphic interface code interacts with the user, and the remote application code manages the sounds, implements all operations, and interfaces with devices. Currently, SoundWorks distributes only the user interface to the sound files and hardware. However, the architecture of SoundWorks can be extended to let multiple users access shared sound files and digital sound hardware throughout the network.

System overview

This section presents the different types of sounds and window interfaces provided by SoundWorks and the operations that modify these sounds.

Sounds and sound windows. SoundWorks provides two types of sound: *sampled* and *generated*. Sampled sounds are stored on disk as a series of numbers representing the change in amplitude over time, while generated sounds are computed as required by software or external hardware. Sounds are accessed through *file sound windows* and *wave sound windows*. A third type of window, the *line segment window*, is used to modify other sounds. Line segment windows allow users to define linear functions. A linear function is a software-generated sound of only one cycle.

In the remainder of this article, we classify all three windows as sound windows. Each type of window supports a common set of controls and operations. Each type also has unique controls and operations. An example of a common operation is setting edit markers that delimit a section of sound or a line in a window. In contrast, the save operation, which saves a sampled sound on disk, is available only for file sound windows, even though line segments could also be saved.

File sound windows are used for accessing sound files, which are stored on disk as a series of samples. Figure 1 shows two example file sound windows. The sounds represented in these windows are identified by name, starting point, and duration. Figure 1a shows a mono sound, and Figure 1b a stereo sound. Sound files can have different sample rates and can be any length in duration. Users can create sounds in a variety of ways. They can sample sounds from an external analog audio source, compute sounds with a direct synthesis language, or create sounds with a SoundWorks operation.

Common editing commands, such as deleting and repeating sections of a sound and changing its amplitude and duration, are accessed through the file sound window menu. The example in Figure 1a has edit marks (vertical lines perpendicular to the sound) that delimit the section of sound starting at time 0.249878 and ending at time 0.641602 seconds. Figure 2 shows the mono file sound window of Figure 1a with the edit window indicating that the delete operation will be applied to the left section of the sound.

Operations on sound files are made to an internal copy of the sound and do not affect the original version stored on disk. The save operation stores a sound file on disk. Sounds can also be manipulated by using operation windows, which are described in the next section.

Wave sounds, which are basic sound units, can be used as building blocks for creating new sounds. Four types of wave sounds are provided: sine, triangle, sawtooth, and square. Before users can access these sounds, they must specify the type of wave, frequency, amplitude, and duration. In contrast to sound files, wave sounds are not stored on disk, but are computed by the system as required. As a result, wave sounds are not modified

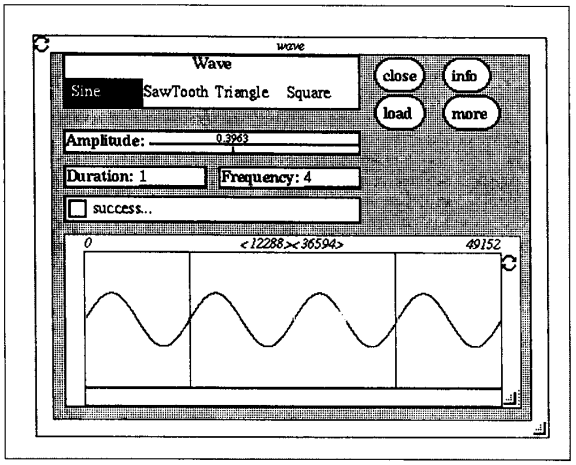


Figure 3. Wave sound window.

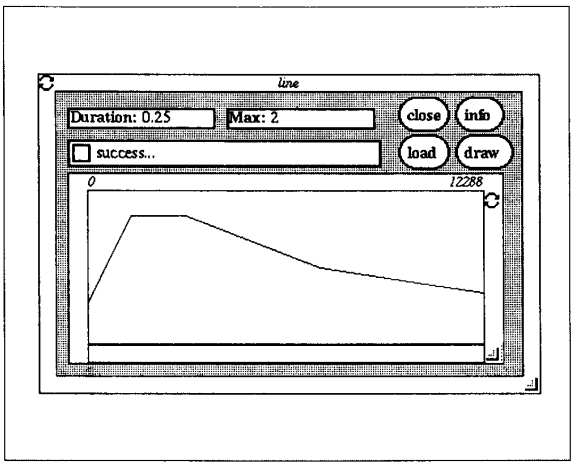


Figure 4. Line segment window.

after they are created. Figure 3 shows an example wave sound window for a sine wave.

A variant of the wave sound generates sound through digital signal processing (DSP) hardware. Instead of a wave type, the user specifies a program executed by the DSP hardware. When the user invokes the program, it generates the sound. For example, a user can specify a line segment window as the waveform shape to produce a wave sound.

Line segment windows are used to create line segments for manipulating sounds. For example, line segments may be used to modify the amplitude and duration of file or wave sounds. Figure 4 shows an example line segment window.

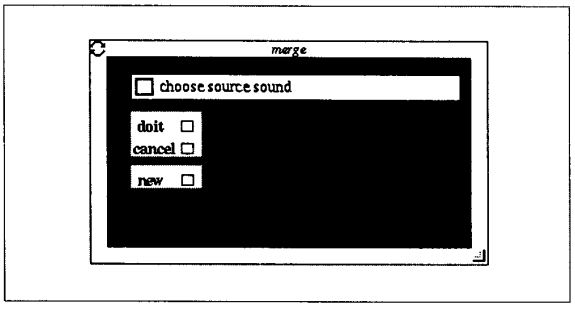


Figure 5. Operation window.

Operation windows. In addition to the editing commands accessed through the menu of a sound window, the SoundWorks system provides a set of operations that uses either a sound or a line segment to manipulate another sound. Users access these operations through separate windows that guide them through the operation, asking them to choose the source, the modifier, and other necessary information. The operation windows can be used to splice and mix sounds as well as to modify the amplitude and duration of a sound by another sound. For example, a line segment can be used as an amplitude envelope to modify a sound. Figure 5 shows a merge operation window at the start of the system's interaction with the user.

Figure 6 shows the input and output windows for a mix operation. The windows in Figures 6a and 6b are wave sound windows. The left one is the source window, and the right is the modifier. Because wave sounds cannot be modified once created, the user must create a new window to contain the result of the operation. To do this, the user chooses the "new" button in the operation window (Figure 5). The window in Figure 6c shows the result of mixing the center section of the source sound with the wave sound and using the result to replace the center section. Currently, SoundWorks supports the use of only two sounds in operations. The capability for supporting any number of these sounds would be a reasonable extension.

SoundWorks design

To design and develop SoundWorks, we used an object-oriented version of PostScript in addition to Sun Microsystems' NEWS application programming environment. Before describing the high-level architecture and the design of the SoundWorks system, we briefly describe the NEWS application environment.

NEWS. Sun Microsystems developed NEWS[6] to provide class structures that support an object-oriented approach for producing graphical interfaces. Through an object-oriented version of PostScript, NEWS provides a set of basic graphic items and an underlying framework for integrating these items into application-specific code. That is, graphic items are defined as a collection of classes, which can be either integrated into the application as is or modified by using inheritance. In SoundWorks we used predefined graphic items as the basis for defining a set of classes for all items in the user interface.

To provide network access to applications, NEWS supports a client-server model for applications development. The term *client server* has an unusual meaning here. The server is the local workstation at which the user's display device resides (in other words, it's a display server), and the client can be the remote system on which the actual application resides. This model allows the application to be partitioned into distinct parts, which can be executed on different systems. The client application interchanges requests with the server at the user's display station. In general, the client application (in SoundWorks

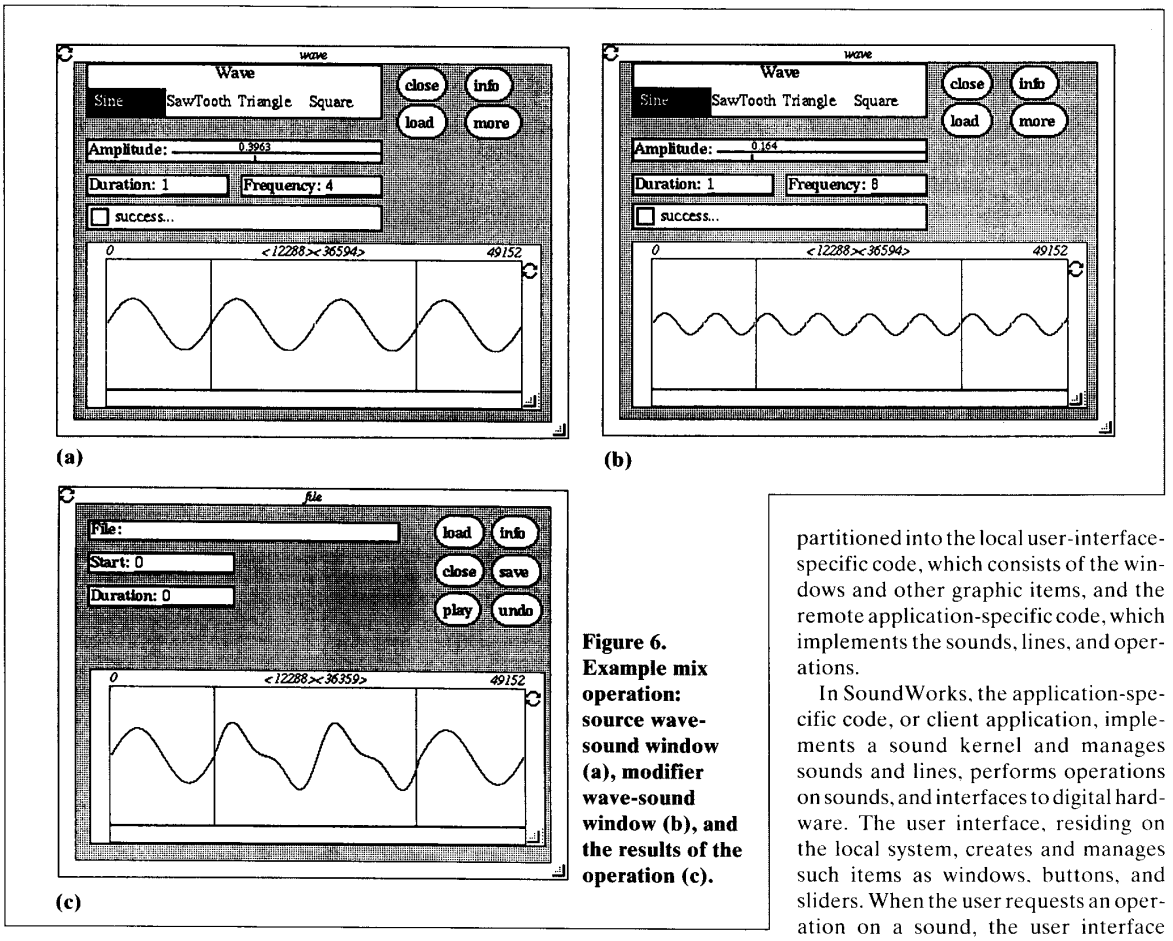


Figure 6. Example mix operation: source wave-sound window (a), modifier wave-sound window (b), and the results of the operation (c).

called the *sound kernel*) deals with the actual application, while the server deals with user interface issues.

To address the performance requirements of interactive applications, NEWS allows application developers to specify their own high-level network protocol to communicate between the application-specific code and the user interface code. The client application downloads the PostScript user interface code to the server at startup. This code thus resides on the server and communicates with the client application code using the network protocol defined by the application developer.

System architecture. We used an object-oriented design approach to develop the high-level architecture for the SoundWorks system. First, we identified the major components of the system and the interactions between them. Next, we partitioned the components for distribution across the network. Class specifications were developed for each component and successively refined into subclass specifications to produce the system implementation.

SoundWorks' major components can be derived from the system overview section. They are the windows that represent sounds, lines, and operations; the sounds themselves; and operations on the sounds. These components can be partitioned into the local user-interface-specific code, which consists of the windows and other graphic items, and the remote application-specific code, which implements the sounds, lines, and operations.

In SoundWorks, the application-specific code, or client application, implements a sound kernel and manages sounds and lines, performs operations on sounds, and interfaces to digital hardware. The user interface, residing on the local system, creates and manages such items as windows, buttons, and sliders. When the user requests an operation on a sound, the user interface communicates with the sound kernel to perform the operation. This communication is completely defined by the protocol between the user interface and the sound kernel. The protocol is implemented by a user interface module and a client application module resident on the server and client systems, respectively. The client application module receives requests from the user interface module and accesses the sound kernel through a well-defined interface. Figure 7 shows a high-level representa-

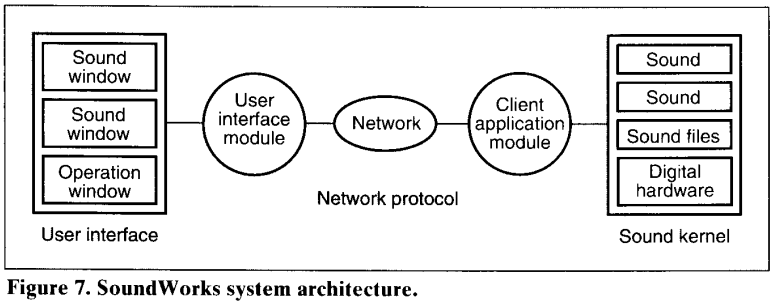


Figure 7. SoundWorks system architecture.

Flow-of-control example

To demonstrate the roles of the objects and modules in the system, we present an example of scaling the amplitude of one sound by a line segment. The example includes the objects that are created and the methods called to perform the operation.

First, a FileSoundWindow must be created and a file sound loaded. To create a FileSoundWindow, the user chooses the create file sound menu command. The creation of the FileSoundWindow and the specification of the file sound are performed on the local system. When the user chooses the load button, the loadSound method first creates a FileSoundView, which in turn creates a new Sound object, one of the classes of the user interface module on the local system. The FileSoundWindow asks the view to load the sound by calling the loadSound method in the Sound object. The Sound object sends a loadFile request to the remote client application. The client application fields the request, performs file_create to create the file sound, and returns the result to the sound object, which returns to the FileSoundView and finally the FileSoundWindow.

If the result is positive, the paint method is invoked to display the file sound in the view. The view invokes the paint method in the Sound object to initiate the drawing process. On receiving the paint request, the client application calls the sound_paint routine in the sound kernel. The sound kernel draws the sound using the paint_window, ps_moveto, and ps_lineto routines provided by the client application. These routines send actual PostScript commands to the user interface to draw the sound. At this point, the sound is visible and can be accessed through the SoundView.

The next steps are to create a LineSegmentWindow and to draw a line segment in it. The LineSegmentWindow is created using the create line menu command. When the user chooses the draw button, the draw method in the LineSegmentView is invoked. The local system handles the drawing of the line segments completely; there is no interaction with the remote sound kernel. The load operation in the LineSegmentWindow invokes the loadLine method in the LineSegmentView, which then calls loadLine in the Sound object. Just as with the file sound, the sound object interacts with the remote client application to create the line and draw it in the LineSegmentView.

At this point, both the file sound and the line segments have been specified; only the scale operation itself needs to be performed. The user chooses the scale operation menu command, which creates a scale operation window. The scale operation window asks the user to choose the source sound to be scaled and the sound (in this case, line segment) to use to scale the sound. This is all done locally without any interaction with the sound kernel. When the user hits the "doit" button, the scaleOperation in the operation object is invoked. This causes a message to be sent to the remote client application. The client application receives the message and calls the scale_operation routine provided by the sound kernel to perform the operation. The result of the scale operation is then returned to the operation object and in turn to the ScaleOperationWindow.

tion of the system architecture. Subsequent sections present the specifications of the user interface, the network protocol, and the sound kernel. An example of the flow of control between the various components of the system appears in the "Flow-of-control example" sidebar.

User interface class specifications. The user interface creates and manages the graphic objects displayed at the user's workstation. We grouped graphic objects that share common features into high-level specifications and used these class specifications to develop subclass specifications for each object of the same type. Examples of objects that share common features are windows, components in the windows, and objects used to represent sounds.

The class specifications for the user interface form a hierarchy based on the class inheritance. Figure 8 presents the inheritance hierarchy for the major classes. At the highest level of the hierarchy are the predefined classes provided by NEWS, including Object, LiteWindow, LiteMenu, and LiteItem. We used the LiteWindow class specification to specify the four types of windows supported by SoundWorks: control, operation, help, and sound. Other windows in

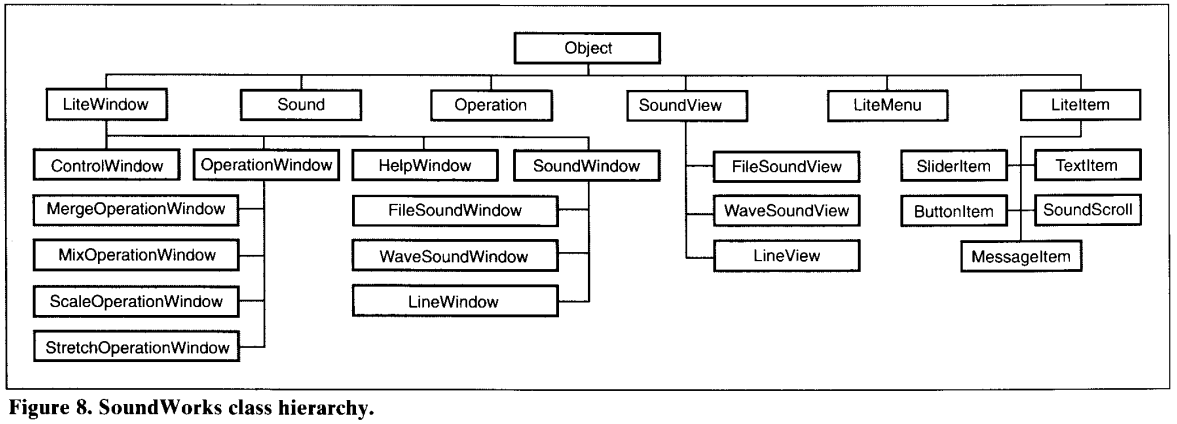


Figure 8. SoundWorks class hierarchy.

SoundWorks are defined as their subclasses.

One class can also be a client of another class. For example, a window is a client of objects like buttons and sliders, and SoundWindows are clients of other classes, including the SoundView class, which is used for displaying and manipulating the different types of sounds. Figure 9a gives the client hierarchy for the FileSoundWindow class, while Figure 9b shows the hierarchy for the MergeOperationWindow class. The other classes have similar client hierarchies.

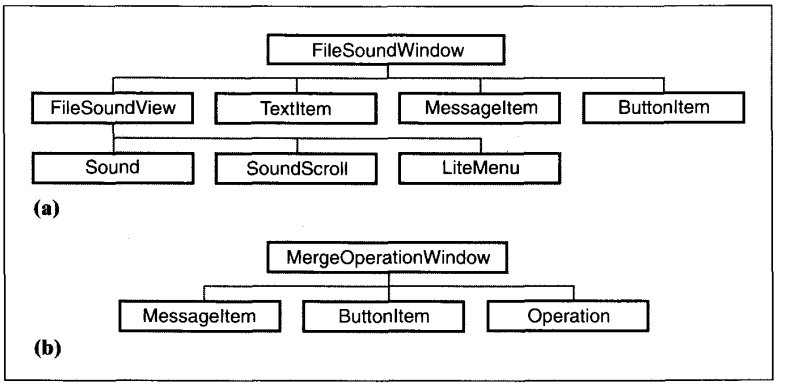


Figure 9. SoundWorks client hierarchy for the FileSoundWindow class (a) and the MergeOperationWindow class (b).

Predefined NEWS classes. At the root of all NEWS classes is the generic class Object (see Figure 8). It defines two methods: "new" and "doit." The new method, which is called to create an instance of the class, is required by every class. The doit method is used to create temporary methods for internal purposes.

NEWS provides a set of subclasses of the class Object that implement windows, menus, and other common items (for example, buttons and sliders). In SoundWorks we used these classes both as superclasses for defining new classes and for clients. Here, we briefly describe the classes provided by NEWS. More details are available in the NEWS user's manual.[6]

LiteWindow. This class provides a standard format for displaying and using windows, and includes a set of controls and menus for manipulating such window features as size and position. LiteWindow defines methods for creating, destroying, moving, and painting the window. For example, the paint method is used to paint or draw the window (and all graphic objects within it).

LiteMenu. This class provides a standard set of pull-right menus that can be associated with a window. A menu is defined as an array of <entry, value> pairs. The entry is the title that will appear in the menu, and the value is either a method to invoke or another menu.

LiteItem. This class provides a set of standard graphic items that can be incorporated into other classes. These items are defined as subclasses of class LiteItem and include items for entering text (TextItem), buttons (ButtonItem), sliders (SliderItem), message areas (MessageItem), and choice items (ArrayItem).

SoundWindow class specifications. This class defines the window used to access sound objects. A subclass of LiteWindow, it inherits the look and functions of that class. The SoundWindow is used as the superclass for a set of subclasses for each type of sound. The SoundWindow class overrides the methods "new" and "paint," as defined in LiteWindow, to include SoundWindow-specific code and to define sound-related methods common to all types of sounds. The methods loadSound, closeSound, and infoSound, which are common to all SoundWindows, are invoked when the user chooses the graphic item (for example, a button or slider) associated with the item (see Figure 1). These methods support a basic interface to the sounds, allowing the SoundWindow to load, close, and retrieve information about sounds, and play file sounds.

The SoundWindow is a client of the SoundView class, which handles the actual display and modification of sounds. (SoundView classes are described in the later section entitled "SoundView class specification.") SoundWindow methods that perform a sound-related operation usually invoke a method in the SoundView class to perform the operation and to display the results. An advantage of this division between SoundWindow and SoundView is that any number of SoundViews may be present in a SoundWindow, thus allowing the display of sound files with any number of tracks.

The specifications we present provide only an overview of the class specification. (For instance, in some cases parameters are missing.) The interested reader can find a complete listing of these specifications provided by Reichbach.[10]

Figure 10 contains the definition of SoundWindow. Each subclass specification overrides the specification for several methods defined in the figure. The createItems method is modified to

```
class:          SoundWindow
superclass:     LiteWindow

instance variables/methods:
  sndid         : SoundWindow identifier
  sndview       : instance of a SoundView
  createItems   : method used to create items

class methods:
  new(parent,id) : create an instance of SoundWindow
  paint         : paint window and contents
  loadSound     : load a sound
  closeSound    : close a sound
  infoSound     : get information about a sound
```

Figure 10. Specification of SoundWindow.

```
class:          FileSoundWindow
superclass:     SoundWindow

instance variables/methods:
  filename      : name of sound file
  start         : default starting point in file
  duration      : duration of sound
  createItems   : create items specific to
                  FileSoundWindow

class methods:
  new(parent,id) : create an instance of
                   FileSoundWindow
  loadSound     : load FileSound
  closeSound    : close FileSound
  playSound     : play FileSound
  saveSound     : save FileSound
  undoSound     : undo the last operation
```

Figure 11. Specification of FileSoundWindow.

```
class:          SoundView
superclass:     Object

instance variables/methods:
  sndid         : sound identifier
  height        : height of SoundView
  width         : width of SoundView
  leftedit      : left edit marker
  rightedit     : right edit marker
  sound         : instance of Sound class
  scroll        : instance of SoundScroll class
  createMenu    : create menu specific to SoundView

class methods:
  new(parent,id) : create an instance of SoundView
  paint         : paint window (and contents)
  closeSound    : close SoundView
  infoSound     : get info about sound in SoundView
```

Figure 12. Specification of SoundView.

```
class:          FileSoundView
superclass:     SoundView

instance methods:
  createMenu    : create a FileSoundView menu.

class methods:
  loadFileSound(sndid,filename,start,duration)
                                : load a file
  saveSound(sndid,filename)     : save FileSound
  playSound(sndid,filename)     : play FileSound
  undoSound(sndid,filename)     : undo operation
  repeatOperation               : repeat a section of
                                  sound
  deleteOperation               : delete a section of
                                  sound
  scaleOperation                : change the amplitude
  stretchOperation              : change the length
  zoomOperation                 : magnify the view
  helpOperation                 : invoke on-line help
```

Figure 13. Specification of FileSoundView.

```
class:          OperationWindow
superclass:     LiteWindow

instance variables/methods:
  op            : operation object
  opid          : operation id
  sound1        : source sound
  sound2        : destination sound
  new?          : create a new sound
  createItems   : method to create items in window

class methods:
  new(parent,id) : create an instance of
                   OperationWindow
  paint         : paint operation window (and
                  contents)
  doOperation   : ask user to choose sounds or lines
                  to operate on
  cancelOp      : cancel an operation and reset
                  variables
```

Figure 14. Specification of OperationWindow.

```
class:          MergeOperationWindow
superclass:     OperationWindow

class methods:
  doOperation   : ask user for information, do merge
```

Figure 15. Specification of the Merge operation.

create subclass-specific items, such as text areas, sliders, and buttons. The new method is modified to create a SoundView for each type of sound. The three subclasses of SoundWindow are FileSoundWindow, WaveSoundWindow, and LineWindow. These are used to access sounds stored on disk, access sounds represented as waveforms, and define line segments, respectively.

FileSoundWindow. This subclass provides access to sounds stored on disk. Before accessing a sound file, the user must specify the variables defining the file name, starting point, and duration. The FileSoundWindow overrides the createItems method to define sound-file-specific items for inputting the name of the sound file and the start and duration times. The loadSound method is modified to invoke the sound-file-specific load method. In addition, the saveSound, playSound, and undoSound methods are defined. The first one saves a sound file on the disk, the second plays a sound

file, and the third undoes an operation. The current version of SoundWorks supports only one level of undo: the last operation.

The FileSoundWindow is defined in Figure 11.

WaveSoundWindow. This subclass helps access sounds represented as simple waveforms. There are five types of waveform sounds: sine, triangle, sawtooth, square, and DSP. Wave sounds require that the frequency, amplitude, and duration be specified before the sound is loaded. This class specification defines the sine, sawtooth, and triangle wave sounds. The square wave sound and DSP sound are specified as separate subclasses because they require additional information. The WaveSoundWindow supports only these basic waveforms because the underlying implementation of wave sounds is in software (which can be slow for long wave sounds). To obtain better performance, we intend to implement more complex wave sounds using the DSP SoundWindow.

WaveSoundWindow classes are specified in a manner similar to the FileSoundWindow and are not shown here. The class specification and more details are provided by Reichbach.[10]

LineWindow. This subclass helps define line segments. It is similar to the FileSoundWindow and WaveSoundWindow, containing methods and variables to load, close, and get information about line segments. Due to space constraints, it is not shown here.

SoundView class specification. This specification defines the graphic representation and operations on sounds contained in the SoundWindow. Corresponding to the subclasses of SoundWindow, there are three subclasses for the SoundView class: FileSoundView, WaveSoundView, and LineView. When a sound or line window is created, an instance of a subclass of SoundView is also created and assigned to the "sndview" variable for the sound or line window. Then, whenever the user requests an operation defined in a window, a method in the SoundView class is invoked to perform the operation. The instance variable "sound" refers to an instance of class Sound (which is presented in the later section entitled "User interface module") and is used to perform sound- and line-segment-related operations. The local instance method createMenu is used to define the specific menu for each subclass. The SoundView class also creates an instance of class SoundScroll, which is used to scroll through the sound displayed in the SoundView. Figure 12 shows the SoundView class definition.

The object-oriented approach to the design of the user interface supported an incremental development.

Like SoundWindows, each subclass of the SoundView class defines the menu and methods specific to the subclass. The menu includes both common operations like zoom and help, and specific operations for each type of sound. In addition, each subclass defines a method to load the correct type of sound (or line). For example, the FileSoundView class defines a method to load the file sound. The loadFileSound method is invoked by the loadSound method in the FileSoundWindow. The FileSoundView class also defines file-sound-specific methods for playing the sound, saving the sound, and undoing the last operation on the sound. Other subclasses define their own menus and methods. For example, the LineView specifies a menu that includes common operations, such as zoom, edit, and help, and specific operations such as drawing line segments.

Figure 13 shows the definition of the FileSoundView class. The WaveSoundView and LineView subclasses are defined similarly.

OperationWindow class specifications. These specifications define windows that can be used to perform operations in which one sound or line modifies another sound. The operations accessed through these windows are merge, mix, scale, and stretch. Each uses the same command syntax and parameters and requires the user to specify the two sounds for the operation and whether a new sound window should be created.

The mix operation adds two sounds together and produces the result. The merge operation inserts the destination sound into the source sound. The scale operation modifies the amplitude of the source sound using the destination (usually a line segment). The stretch operation, implemented using linear interpolation, is like a variable-speed playback, using the destination sound as a rate of change for the source sound. For example, if a constant-valued destination sound (say, a line segment of value 2) is used to stretch a source sound, the result is a sound with double the pitch of the source sound and half the duration.

Operation windows are based on a generic class called an OperationWindow, which specifies the format and usage of the operation. OperationWindows are identified by a unique variable "opid," which identifies each active operation. The OperationWindow creates an instance of the Operation object, "op," which contains operation-specific methods. The OperationWindow corresponding to each specific operation is defined as a subclass of the class OperationWindow and invokes an operation-specific method in the Operation object. Figure 14 shows the specification for class OperationWindow.

Each subclass of OperationWindow invokes an operation-specific version of the doOperation method. Figure 15 shows the specification for the Merge operation; other subclasses are specified in the same manner. The specification for the Operation class is presented in the later section entitled "User interface module."

Protocol specification

The protocol defined between the user interface code and client application is partitioned into two modules. The user interface module defines an interface for each operation and sound supported by SoundWorks. The client application module receives requests from the user interface module, calls the sound kernel to perform the requests, and returns the results to the user interface.

User interface module. This module is defined by two classes: Sound and Operation. The Sound class defines the message format for each sound-related request, while the Operation class defines the message format for each operation. When a method in a Sound object

```
class:          Sound
superclass:     Object

class methods:
  new(parent,id)                                  : create a sound object
  paint(sndid,height,width)                       : paint the sound
  closeSound(sndid)                               : close the sound
  infoSound(sndid)                                : get information
  saveSound(sndid,filename)                       : save sound file
  playSound(sndid,filename)                       : play sound file
  undoSound(sndid,filename)                       : undo operation
  loadFile(sndid,filename,start,duration)         : load sound file
  loadWave(sndid,type,freq,amp)                   : load a wave sound
  loadLine(sndid,points)                          : load a line segment
(a)

class:          Operation
superclass:     Object

class methods:
  new(parent,id)                                  : create an operation object
  mergeOperation(opid,sound1,sound2,new?)         : merge sounds/lines
  scaleOperation(opid,sound1,sound2,new?)         : scale sounds/lines
  stretchOperation(opid,sound1,sound2,new?)       : stretch sounds/lines
  mixOperation(opid,sound1,sound2,new?)           : mix two sounds/lines
(b)
```

Figure 16. Specifications of the Sound (a) and Operation (b) classes.

```
/* Client Application Module */
loop
    wait for request
    decode the request
    call sound kernel
    /* sound kernel calls routine to return result */
until user is done
```

Figure 17. Client application module.

```
sys_message(string)                       : display a system message for user
error_message(cid,string)                 : display a message in a window
set_info(cid,size)                        : return the size information
return_result(cid,result)                 : return a result
return_result_string(cid,result,string)   : return a result and string
paint_window(cid)                         : start to draw a sound in window
paint_icon(cid)                           : start to draw a sound icon
ps_moveto(x,y)                            : move to (and set current position) x,y
ps_lineto(x,y)                            : draw a line from current position to
                                            x,y
```

Figure 18. Routines provided by the client application module.

or an Operation object is invoked, a message is sent to the client application module. After sending this request, the calling object waits for the result. Figures 16a and 16b show the specifications for the Sound and Operation classes.

Client application module. This module provides the interface between the objects in the user interface and the sound kernel. As with other NEWS-based applications, the client application module is written in C and uses the libraries provided by NEWS to communicate with the user interface code.

As Figure 17 shows, the main functions of the client application module are to wait for a request from the user interface module, decode the request, and perform it by calling the sound kernel.

The client application module also provides a set of routines called by the sound kernel to return the result of each request. These routines communicate with the user interface to return result values (with or without an accompanying text string, like an error message), to return size information, and to paint the graphic representation of the sound (or line). They provide access to the user interface and are independent of the application programming environment. This allows other user interfaces, using possibly other application programming environments, to be integrated with the sound kernel in a straightforward manner. The integration effort requires specifying the above routines, writing the client application module loop, and mapping user interface requests to the sound kernel interface.

For example, to integrate the sound kernel with an X Windows-based application would require the X client application to call the sound kernel routines (defined in the next section) in response to user requests. In addition, the X Windows application would have to provide the client application routines that allow the sound kernel to return the result of the operation to the application. For example, the routine error_message would display a message for the user, while ps_moveto would map into the equivalent X routine to draw a line. Figure 18 lists the routines provided by the client application module for communicating with the user interface. When the sound kernel finishes performing a request for the user, these routines return the result.

Sound kernel specification

This kernel manages sounds and lines, performs operations on sounds, and in-

terfaces to the digital hardware. These functions implement the methods defined in the Sound and Operation classes. In response to user requests, the client application module accesses the functions through the interface described in the next section. The sound kernel interface is independent of the client application module interface that accesses it. The sound kernel accesses the routines supplied by the client application module to return the result of a request, report errors, and draw the graphic representation of a sound or line.

Interface to sounds. The sound kernel manages three different types of sounds: sound files, wave sounds, and lines. Each sound provides a unique create routine used to create the sound, as well as generic sound routines common to all sounds. Once a sound is created, it is accessed in the same manner regardless of its type.

Sound-specific routines. The routines shown in Figure 19a create an internal structure for a sound of the specified type. The info parameter provides specific information, such as the sample rate and sample size. Once created, a sound is loaded using a load operation (see Figure 19b) that inserts the requested sound into the previously created sound structure. For example, the load_file operation loads the sound file specified by filename, starting time, and duration into the sound structure. The amount of sound that can be loaded is limited only by the size of the address space of the underlying computer.

Generic sound routines. These routines access the sound and are called in response to user requests. Currently, SoundWorks supports the sound_write and sound_save routines only for file sounds. The sound_write routine writes the specified portions of the sound, while the sound_save routine saves the entire sound on the sound file system. Figure 20 shows the routines.

Interface to operations. The sound kernel provides the routines shown in Figure 21a for each operation supported by SoundWorks. The operations access the sounds through the interface defined in the previous section and return operation results by using the routines supplied by the client application module. The parameter id indicates the operation window that requested the operation. The parameters sound1 and sound2 identify the source and destination sounds. If the parameter newflag is set, then a new file sound is created as a result of the operation.

FileSounds also have the two operations shown in Figure 21b to delete and repeat sections of sound. The left and right parameters delimit the section of sound to operate on.

Interface to digital hardware. The sound kernel is also responsible for interfacing with the digital hardware: the sound file system, play/record devices, and the DSP hardware.

The interface to the sound file system depends on which system is used. SoundWorks was designed using the CARL (Computer Audio Research Laboratory) sound file system developed for Unix.[1] To access the sound files and to play or record them, SoundWorks relies on the libraries and commands provided by the system. Commands are specified either in the user's environment or in a SoundWorks configuration file.

In addition, SoundWorks provides access to the AT&T VMEbus DSP32 signal-processing module.[11] This allows the user to integrate digital signal processing programs into the SoundWorks environment in a straightforward manner. Using this interface requires knowledge of the DSP32 module and the interface between the

```
file_create(sndid,info)  : create a file sound
wave_create(sndid,info)  : create a wave sound
line_create(sndid,info)  : create a line sound
(a)

load_file(sndid,filename,start,duration)
load_wave(sndid,type,frequency,amplitude,duration,
  extra)
load_line(sndid,points)
(b)
```

Figure 19. Sound-specific routines to create an internal sound structure (a) and operations to load sounds (b).

```
sound_close(sndid)                 : close the sound
sound_info(sndid)                  : return information about the sound
sound_paint(sndid,height,width)    : paint the sound
sound_read(sndid,start,duration)   : read samples from the sound
sound_write(sndid,start,duration)  : write samples to the sound
sound_save(sndid,name)             : save the sound
```

Figure 20. Generic sound routines.

```
merge_operation(id,sound1,sound2,newflag)  : merge sounds
scale_operation(id,sound1,sound2,newflag)  : change the amplitude
stretch_operation(id,sound1,sound2,newflag) : change the duration
mix_operation(id,sound1,sound2,newflag)    : mix two sounds
(a)

delete_operation(id,sound1,left,right)     : delete section of sound
repeat_operation(id,sound1,left,right)     : repeat section of sound
(b)
```

Figure 21. Routines to perform operations on sounds (a). Delete and repeat operations (b).

DSP programs and internal sound data structures.

Achievements

We developed a working implementation of SoundWorks that runs on our network of Sun workstations. SoundWorks is also installed at the Music Department's Center for Computer Music Composition, where the sound kernel is resident on a Digital Equipment Corp. VAX 11/750 and a Digital Sound Corporation digital-analog converter, with a Sun computer serving as the graphics workstation. A third installation, in the Electrical and Computer Engineering Department's Communications Research Lab, is based completely on Sun workstations and includes an AT&T VMEbus DSP32 signal-processing module.

We achieved both goals set for the SoundWorks system: sound manipulation features and a flexible server-based system. The system provides all of the sound manipulation and creation features presented in the "System overview" section except the DSP window. Integrating the user interface with the sound kernel demonstrated the viability of the server-based architecture.

Experience with NEWS. The development of SoundWorks gave us firsthand experience in using the NEWS application programming environment and the object-oriented version of PostScript. The object-oriented approach to the design and implementation of the user interface supported an incremental development of the system. This let us develop and refine components in a structured manner.

The support for distributed applications was particularly useful. The division of the user interface and the application code made design and development easier because we could develop each section independently.

Problems encountered with using NEWS fall into two categories: difficulties with the object-oriented version of PostScript and problems with NEWS itself. PostScript, while very good at graphics, is not a well-structured programming language. For example, all parameters are passed on the stack and there is no explicit checking for the number of parameters passed or for their type. And because NEWS was a new product, it had its own set of bugs and problems.

An on-line help facility provides information on every sound and operation.

Experience with SoundWorks. SoundWorks provides a user-friendly interface for manipulating sounds. Because the graphic interface supports a consistent user interaction regardless of the type of sound, learning to use SoundWorks is easy. In addition, the on-line help facility provides information on every supported sound and operation.

Some problems did arise from the lack of accuracy in performing certain editing operations. In particular, operations that involve a small number of samples are difficult to perform because the graphic representations are not accurate enough. Of course, the user can zoom a sound to the necessary level of detail, but this requires extra commands.

There are also minor annoyances in the user interface code. These are mainly in the interface defined for choosing the sounds and sections of sound when using operation windows. Currently, the operation window fixes the order of sound choice and does not tolerate user error. A mistake forces the user to restart the operation. In addition, when all parameters have been input and the operation is started, it is impossible to halt the operation before completion. A related issue is the lack of support for grouping operations together so that the user can specify common operation sequences by using a macro language.

In addition, drawing performance for large sounds needs to be addressed. For example, on a Sun-3 it takes approximately 5 seconds per minute of sound to compute and display the graphic representation of the sound. Almost all of the time required to display a sound is taken up in two tasks: computing the bitmap representing the sound and the actual graphics operations themselves. One method to reduce the time needed for producing bitmaps is to generate intermediary bitmaps. This reduces the amount of data processed to generate the required bitmap. Also, to reduce the network overhead of drawing the bitmap, it is possible to compress the amount of data transferred over the network, thereby increasing performance.

The SoundWorks system provides a good starting point for future work. Two possible directions are the integration of new applications and the distribution of the sound kernel architecture. For example, it would be desirable to have different users, possibly running different applications, accessing the sound kernel concurrently. Distributing the sound kernel would allow users to access audio devices and sound files residing on other workstations. An approach we are considering for accessing the audio devices at another workstation is the integration of the sound kernel with a Vox server.[9] We could also distribute the sound file system itself on different machines.

Another promising area for further research is the adoption of a sound file system that supports nondestructive editing to the SoundWorks kernel. This type of editing lets users "modify" sounds without actually modifying them. For example, an insertion is accomplished by adjusting pointers to the sound, not by changing the sound itself.

It would also be desirable to port SoundWorks to other architectures. One candidate is the Next computer, which has integrated digital audio hardware. However, because the Next computer's window system is not distributed, additional work would be required to develop a network interface between the user interface and the sound kernel. ■

References

1. *CARL Music Manual*, Department of Music, University of California, San Diego, 1984.
2. A. Freed, "MacMix — Mixing Music with a Mouse," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1986, pp. 127-129.
3. M. Lentczner, "Sound Kit — A Sound Manipulator," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1985, pp. 141-145.

4. H. Lieberman, "Machine Tongues IX: Object-Oriented Programming," *Computer Music J.*, Vol. 6, No. 3, Fall, 1982, pp. 8-21.

5. G. Loy and C. Abbott, "Programming Languages for Computer Music Synthesis, Performance and Composition," *ACM Computing Surveys*, Special Issue on Computer Music, Vol. 17, No. 2, June 1985, pp. 235-265.

6. *NEWS 1.1 User's Manual*, Sun Microsystems, Mountain View, Calif., 1987.

7. S. Pope, "Building Smalltalk-80 Based Computer Music Tools," *J. Object-Oriented Programming*, Vol. 1, No. 1, Apr./May 1988, pp. 6-10.

8. D. Terry and Daniel Swinehart, "Managing Stored Voice in the Etherphone System," *ACM Trans. Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 3-27.

9. B. Arons et al., "A Breadboard Model for the Vox Server," draft, Olivetti Research, Menlo Park, Calif., June 1988.

10. J. Reichbach, *SoundWorks: A Distributed System for Manipulating Digital Sound*, master's thesis, University of California, Santa Barbara, 1989.

11. H. Alrutz, "Using the VMEbus DSP32 Signal-Processing Module in a Unix Environment," Tech. Memo 11224-861031, AT&T Bell Laboratories, Murray Hill, N.J., 1986.

Jonathan D. Reichbach does research in digital audio and object-oriented systems design, which is applied to developing professional digital audio systems for the recording, film, and television industries. He continued to work on SoundWorks while at Sun Microsystems, initiating the integration of a new sound file system (the MultiMedia File System), which supports nondestructive editing and graphics performance improvements. He also used his SoundWorks experience to develop the Sonic Solutions SonicSystem, which is used in the recording industry for CD premastering applications, and film and video audio production.

Reichbach received his BS in computer science from the University of New York at Stonybrook in 1979 and his MS in computer science from the University of California at Santa Barbara in 1990.

Richard A. Kemmerer is a professor in the Department of Computer Science at the University of California at Santa Barbara. He has been a visitor at MIT, Wang Institute, and Politecnico di Milano. From 1966 to 1974 he worked as a programmer and systems consultant for North American Rockwell and the Institute of Transportation and Traffic Engineering at UCLA. His research interests include formal specification and verification of systems, computer system security and reliability, and computer music.

Kemmerer received his BS in mathematics from Pennsylvania State University in 1966, and his MS and PhD in computer science from the University of California at Los Angeles in 1976 and 1979. He is a senior member of the IEEE, a member of the IEEE Computer Society, a member of the International Association for Cryptologic Research, past chair of the IEEE Technical Committee on Security and Privacy, and a member of the advisory board for ACM's Special Interest Group on Security, Audit, and Control.

Readers can contact Richard Kemmerer at the University of California at Santa Barbara, Computer Science Department, Santa Barbara, CA 93106.