Mapping data and audio using an event-driven audio server for personal computers

Michael Hamman
University of Illinois Urbana-Champaign
705 W. Nevada # 4
Urbana, IL 61801
m-hamman@uiuc.edu

Camille Goudeseune
University of Illinois Urbana-Champaign
603 W. Nevada #1
Urbana, IL 61801
cog@uiuc.edu

ABSTRACT

Recent research suggests that auditory display offers new means for observing and differentiating complex data. One standard method for rendering an auditory display is playback of previously generated audio files. This method is enhanced through playback modification using tools such as Intel's RSX. MIDI-based synthesizers provide yet another method for auditory display. These methods have various drawbacks that are pressed to the limit when confronted with the requirements of analogical display systems. What is needed, therefore, is a way of rendering audio that is to auditory display what a system like OpenGL is to graphical display. Audio Rendering Engine And Library (AREAL) is a real-time audio renderer and sound synthesis software library. It offers the software developer and auditory display designer a set of tools for developing high-quality audio applications for low-cost multimedia computers using consumer or professional audio hardware. The primary purpose of AREAL is to enable a "model-based" approach to audio which is becoming common on high-end (and high-cost) workstations. This paper gives a brief description of this software system and its potential for use within the auditory display community.

INTRODUCTION

Recent research suggests that auditory display offers new means for observing and differentiating complex data [3][7]. An auditory display can be used to signal the occurrence of discrete events or to signal the state of an evolving multivariate data set [1][3][8]. This leads to a variety of approaches. At one extreme, auditory events are associated to data *symbolically*; that is, their meaning is characterized in terms of the source to which they are attributed [6][7][8]. At the other extreme, auditory events are associated with data *analogically*; that is, their meaning is characterized in terms of the interrelations within and among data [8][10].

One standard method for rendering an auditory display is playback of previously generated audio files. An advantage of this method is that it allows for precisely controlled studio engineering involving recording, synthesis, and processing techniques that would be computationally prohibitive for real-time rendering. In addition, tools for editing and playback of audio files are readily available under virtually any operating system. A disadvantage of this method is that multiple playback of the same audio file, over and over, can become tiresome to the ear. Moreover, while disk storage generally costs little, the management of large numbers of audio files is unruly and inevitably error-prone. Finally, no matter how many different audio files are used, there will always be holes in the mapping of data features to acoustic features if the data or process to be mapped is even slightly complex. This becomes more and more the case as we move toward analogical displays.

One way to deal with this situation is to employ already existing (and for the most part, free) audio tools which allow for real-time manipulation of stored sound files. This can be a viable solution for "symbolic" auditory displays representing discrete events and process states. Through the manipulation of a single sound file, one could represent many different states of a display object such as an icon or mouse cursor. However, this approach still suffers under the requirements called for with analogic displays. In order to deliver effective analogic representations, an audio rendering system is needed which can be deeply mapped to the model being represented.

At first glance, MIDI-based synthesizers offer an appealing alternative to the use of sound files. Using MIDI, a synthesizer can be triggered by mapping data to particular MIDI messages [7]. The problem with MIDI, however, is two-fold. First, one must accept the sounds that are given by the particular synthesizer technology at ones disposal. These sounds may differ, depending on the particular device being used. Second, MIDI is limited by its narrow control bandwidth and thus cannot be used for the display of highly multivariate and rapidly evolving data [9]. This has a secondary consequence which is that while one might solve the problem of mapping multivariate data to sound through the use of system exclusive messages (messages that are manufacturer-specific), these messages exacerbate the problem of narrow bandwidth since they usually require higher bandwidth per event.

What is needed is a way of rendering audio that is to auditory display what a system like OpenGL is to graphical display. Moreover, tools are needed with which auditory display designers can test hypotheses and experiment with different methods by which a data model might be rendered acoustically.

AREAL

Audio Rendering Engine And Library (AREAL) is a real-time audio renderer and sound synthesis software library. It offers the software developer and auditory display designer a set of tools for developing high-quality audio applications for low-cost multimedia computers using consumer or professional audio hardware. The primary purpose of AREAL is to enable a "model-based" approach to audio which is becoming common on high-end (and high-cost) workstations. We understand "model-based" audio to be that in which auditory events are generated in real-time as a direct mapping of a dynamically evolving data set or process.

AREAL incorporates (1) a real-time scheduler for managing uninterrupted playback of computed audio samples using a standard sound card; (2) an extendible synthesis library, whose API is called directly by an application; and (3) a method by which audio rendering "models" are specified.

SOFTWARE ARCHITECTURE

AREAL consists of three software layers: the Scheduler, the Synthesis Engine, and a C-language "wrapper" (figure 1).

The Scheduler handles the passage of samples to the sound card. It does this by iteratively filling buffers with samples as these are computed within the Synthesis Engine and inserting these into the sound card's queue. It maintains as low a latency as possible while providing sufficient safety from audio interruptions due to other processes needing the CPU. When the Scheduler needs more samples, it executes a callback against the Synthesis Engine.

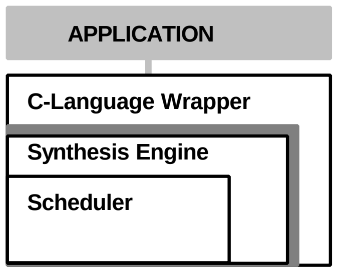

Figure 1

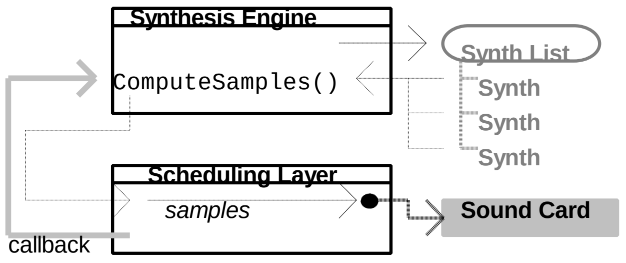

Figure 2

The Synthesis Engine contains a hierarchy of sound synthesis classes and a facility for handling messages from the Wrapper and Scheduling layers. Upon initialization, a list of sound objects is instantiated. When the Scheduler requests samples (through execution of its callback procedure), the Synthesis Engine obtains buffers of samples from each synthesis object and sums them into the buffer that has been passed from the Scheduler (figure 2).

The sound synthesis class library is a collection of C++ classes, each of which specifies a particular synthesis algorithm and defines a standard class interface. We envision the sound synthesis class hierarchy as an "open" architecture, enabling addition of customized synthesis algorithms and extensions of current algorithms based on particular rendering and optimization requirements. Currently, the library is extendible only at the source-code level. In a future version, however, it is anticipated that this extendibility will be realized through implementation as an ActiveX container. ActiveX is a Microsoft construct that allows different applications to exchange information and for one application to embed itself into the other, thus becoming, in effect, a component of it. As ActiveX controllers, sound synthesis libraries could literally be "dropped into" AREAL and thus become one of its synthesis libraries. This kind of extendibility holds great promise for the future of audio applications in general and for auditory display in particular.

Sample underflow (which can cause annoying clicks in the audio signal) is handled through a facility called the *OctaneMeister*. The OctaneMeister anticipates possible sample underflow by monitoring CPU load. When this load nears 100 per cent, the OctaneMeister tells the list of instantiated synthesis objects to lower its computation "octane." Each synthesis object has its own method for reducing its octane which is defined within the class from which it is instantiated. Moreover, synthesis objects are ranked in order of complexity: those whose performance would be most seriously compromised through the lowering of its "octane" are placed at the bottom of the list.

The C-language "wrapper" forms a shell around the Scheduler and the Synthesis Engine. It exports a set of C functions to the client application. This set of functions is referred to as the "API." The API constitutes the sole interface between the application and AREAL (as depicted by the single bar connecting the APPLICATION and the Wrapper in figure 1). The motivation for defining the interface as a C-language API, rather than allowing the application to interface directly to the C++ library classes within AREAL, is to make the interface as simple as possible to use and to allow for applications written in languages other than C++ to be linked to the library. Functions defined within the API instantiate synthesis objects, set up of data-to-renderer mappings, and handle control messages to individual synthesis objects.

MAPPING DATA TO AUDIO FROM WITHIN AN APPLICATION

Finding flexible though coherent methods for mapping data to an audio-rendering agent is a challenging task. Our approach is predicated on the desire that once such a mapping has been established, an application should no longer have to think about how its data and processes are being acoustically rendered. Therefore, each time there is a change made to a variable which is mapped to an audio parameter, a call is made to a function within the AREAL API, telling AREAL the new value of that variable. This value is then dispatched to the appropriate synthesis algorithm.

Within the application, each data point that is to be rendered is mapped to a parameter within a sound synthesis algorithm. Such an algorithm is encapsulated in a synthesis *class* residing within AREAL. A sound synthesis parameter is used to control some aspect of the synthesis algorithm. In a relatively simple case, such a parameter could control the frequency of a single sine tone, while another could control its amplitude. In more complicated cases, such a parameter might control some aspect of a more elaborate synthesis algorithm.

In an effort at a clear explication of how this works, a simple example is provided. In this example, we start with a data set that contains two variables, x and y, which describe a Cartesian graph. We select a simple additive synthesis algorithm which is defined by 3 parameters: frequency (F), amplitude (A), and number partials (T). Within the application (with which the AREAL library has been linked), first a range is defined for each synthesis parameter. In our example, F will have the range [100Hz, 1100Hz], A will have the range [10000, 20000], and T will have the range [0 partials, 20 partials].

After ranges have been defined for each synthesis parameter, a single C-language function call is made to set the mapping between these data points and synthesis parameters:

```
SetMapping(hSynthObj,"F=(x,10,40); A=.3*(x,10,40)+.7*(y,.03,-.03); T=(y,.1, .9)");
```

`SetMapping()` has two arguments: a pointer (handle) to the synthesis object (which has been previously defined within the application), and a string which specifies the mapping between application data points and synthesis parameters. This string is parsed within the AREAL library in order to realize the specified mapping. In the above example, the two data points are mapped to the three synthesis parameters as follows. All synthesis parameters in the mapping are normalized to the range [0,1] relative to the range set earlier in the program. `F` is a direct linear mapping from data point `x`: as `x` increases from 10 to 40, `F` increases from 0 to 1. `A` is a weighted sum of data points `x` and `y`, 30% and 70% respectively, with `x` traversing the domain (10, 40) and `y` traversing the domain (0.03, -0.03). So, for instance, if `x`=25 and `y`=0.0, then `A` would have a normalized value of (.399=(.3 * .33)+(.7*.5)). Meanwhile `T` maps directly to data point `y` with the domain (.1, .9).

Such a mapping is specified once within the application. After this, whenever the state of one of the data points is changed a call is made to the AREAL API to inform it of this change in state:

```
...
x = getXValue();
SetParmValue(hSynthObj,"x", x);
...
```

Given a sequence of states for x and y, a correlated sequence of parameter states is generated within the synthesis algorithm. Such a situation is shown in figure 3, in which a sequence of 5 states for x and y is correlated to a similar sequence for synthesis parameters F, A, and T. These synthesis parameters are in turn mapped against the range with respect to which they have been defined for this application. The values for the synthesis parameters thus realized are shown in figure 4. As can be observed, this sequence defines an acoustical behavior in which frequency rises from 100 to 370 Hz, while amplitude drops from 17000 to 15400 and the number of partials goes from 5 to 0 partials. The resultant behavior defines an acoustical "gesture" which, while rising in pitch, descends in loudness and in timbral richness. This "gesture" constitutes a potentially informative and evocative representation of the data it renders in which x rises continuously while y descends somewhat sharply before coming back up again slightly.

x=10	y= .3	→	F=0.0	A=.7	T=.25
x=12	y= .2	→	F=0.067	A=.6	T=.125
x=14	y= .1	→	F=0.13	A=.51	T=0.0
x=16	y= .0	→	F=0.2	A=.41	T=0.0
x=18	y= .1	→	F=0.27	A=.54	T=0.0

Figure 3

F=100Hz	A=17000	T=5 partials
F=167Hz	A=16000	T=2 partials
F=230Hz	A=15100	T=0 partials
F=300Hz	A=14100	T=0 partials
F=370Hz	A=15400	T=0 partials

Figure 4

***AUDIO DESCRIPTION* FILES**

Mapping data to audio parameters from within the application itself has one drawback: if one wishes to change the mapping, then the application has to be recompiled. Not only is this burdensome for auditory display designers who have access to the source code, it can prohibit those who are not themselves programmers from being able to make changes in that mapping at all.

Audio description files define data-to-renderer maps within a text file. The application reads this file during its initialization, and the mapping designated within that file is applied inside the application. Designers and composers can then effect changes in data-to-rendering mapping by making changes with respect to this file.

DEFINING AN AUDITORY DISPLAY OF LARGE DATA SETS

Such an organization can be used to represent an arbitrarily large data set whose values unfold at any number of rates. Imagine, for instance, a data set involving three Cartesian maps, such as the one described above, plus additional data points. Figure 5 depicts the patch with which the corresponding acoustical representation might be defined. Each of the three Cartesian sets are shown as Sets A through C. Synthesis objects are labeled Synth1, Synth2, and Synth3. As shown, Cartesian set A is mapped to synthesis object 1, and so on. Meanwhile, other data points (grouped together, for the sake of illustration, as "Other data points") are mapped to parameters by which a reverberation processor unit is controlled and by which channel placement is controlled.

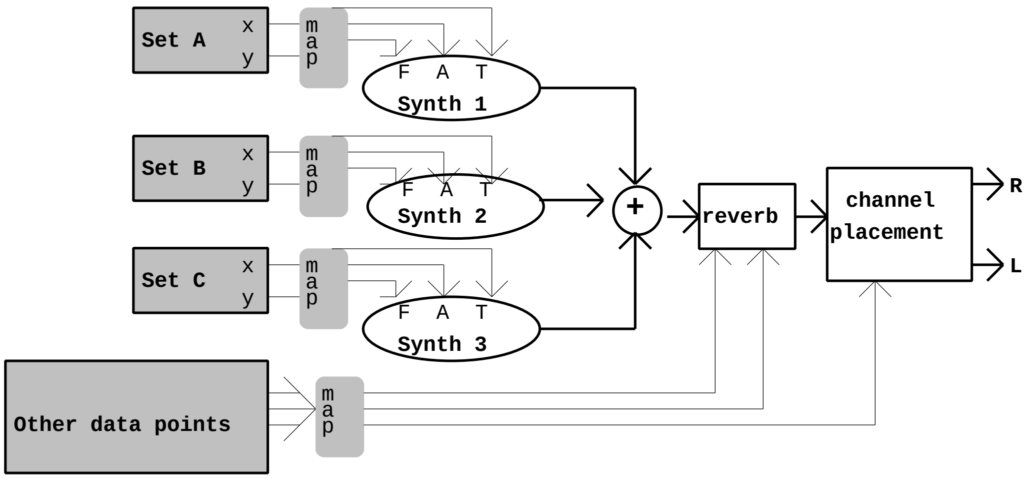

Figure 5

The resulting signal reads from left to right, beginning with the additive synthesis objects. Samples produced by each of the synthesis objects are summed, and placed through the reverberation processor. This reverberation processor, has for the sake of simplified illustration, two parameters: reverberation time and delay time. The resulting samples are then passed to the channel placement module, where each is multiplied by the value defined by that module's single control parameter. For each sample, this value is placed in the right-channel buffer cell, while its inverse value is placed in the left-channel buffer cell.

With this particular arrangement, we are able to map a 12-D data set to elements of an auditory display. Using an Audio Description file, a designer can fine-tune this mapping until s/he finds the configuration which best reflects an informative view of the data being represented.

FUTURE DEVELOPMENTS AND CURRENT WEAKNESSES

AREAL proposes a possible solution to the "build it yourself" problem which currently plagues research and development in auditory display. As such a proposal, it is in the infant stage of a potentially viable technology. As such, its future development is dependent on feedback from other researches within the auditory display community. Currently, we are planning to develop AREAL as an ActiveX server. We are also planning on adding network capabilities so that AREAL can act as a server in a network environment. Thus, a low-cost NT workstation could serve as the audio server, while the applications whose data it renders can run on other computers. We are also hoping to develop a Netscape plugin so that auditory displays may be rendered over the Web. Finally, we wish to develop graphical tools for specifying and investigating data-to-renderer maps.

REFERENCES

1. Blattner, M.M., Papp III, A. L., Glinert, E. P. "Sonic Enhancement of Two-Dimensional Graphics Displays." in *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ed. G. Kramer. Reading, Mass: Addison-Wesley Publishing Co. 1994.
2. Brewster, S. A., Wright, P. C., Edwards, D. N. "A Detailed Investigation into the Effectiveness of Earcons." in *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ed. G. Kramer. Reading, Mass: Addison-Wesley Publishing Co. 1994.
3. Fitch, W. T., Kramer, G. "Sonifying the Body Electric: Superiority of an Auditory over a Visual Display in a Complex, Multivariate System." in *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ed. G. Kramer. Reading, Mass: Addison-Wesley Publishing Co. 1994.
4. Gaver, W. W. "The SonicFinder: An Interface that Uses Auditory Icons." *Human-Computer Interaction* 4(1): 67-95. 1989.
5. Gaver, W. W., and Smith, R. B. "Auditory Icons in Large-scale Collaborative Environments." in *Human-Computer Interaction - INTERACT '90*, ed. D. Diaper. North-Holland: Elsevier Science Publishers. 1990.
6. Gaver, W. W. "Using and Creating Auditory Icons." in *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ed. G. Kramer. Reading, Mass: Addison-Wesley Publishing Co. 1994.
7. Kramer, G. and, Ellison, S. "AUDIFICATION: The Use of Sound to Display Multivariate Data." *Proceedings of the 1991 International Computer Music Conference*, Montreal. San Francisco: Computer Music Association, 1991. pp. 214-221.
8. Kramer, G. "An Introduction to Auditory Display." in *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ed. G. Kramer. Reading, Mass: Addison-Wesley Publishing Co. 1994.
9. Moore, F. R. "The dysfunctions of MIDI," *Computer Music Journal* 12(1): 19-28. 1988.
10. Sloman, A. "Afterthoughts on Analogical Representations." in *Readings in Knowledge Representation*, ed. R. Brachman and H. Levesque. Los Altos, CA: Morgan Kaufman.