An Organization For High-Level Interactive Control of Sound

Sumit Das
Tom DeFanti
Dan Sandin

Electronic Visualization Laboratory
University of Illinois at Chicago
Department of Electrical Engineering and Computer Science

Abstract

The state of computer generated sound has advanced rapidly, and there exist many different ways of conceptualizing the abstract sound structures that comprise music and other complex organizations of sound. Many of these methods are radically different from one another, and so are not ususally used within the same system.

One problem that almost all methods share is one of control, as large amounts of data are needed to specify sounds. How do we create, examine, and modify these complex structures? The problem is exacerbated if we consider the realm of interactively controlled sound.

This chapter presents an organization which, rather than forcing a particular way of thinking about sound, allows multiple arbitrarily high-level views to coexist, all sharing a common interface. The methods or algorithms are abstracted into a objects called *auditory actors*. This encapsulation allows different algorithms to be used concurrently.

All communication with and between these actors is carried out through message-passing, which allows arbitrary types of information (such as other messages) to be easily communicated. This standardizes control without limiting it to a particular type of data.

A prototype system was implemented using this model. This system was used by a number of different developers to create audio interfaces for interactive virtual reality applications, which were demonstrated at the SIGGRAPH 94 conference in Orlando, Florida. Compared to earlier systems, developers were able to create more complex audio interfaces in a shorter time.

1 Introduction

1.1 Sound in Interactive Applications

Many researchers have created systems, languages, and paradigms for creating sound with the aid of computers [1, 2, 3]. In the past, the utility of these systems for real-time work has been limited to either using very simple algorithms or controlling special-purpose hardware [4, 5, 6].

For many purposes, MIDI-compatible synthesizers suffice for the sound generation hardware. However, these devices tend to be limited, both in the range of sounds of any one unit, and especially in the area of interactive control. Only very recently has the computing power of general-purpose computers become equal to the task of producing reasonably complex sound with good fidelity in real-time.

Given hardware and software capable of generating the types of sounds desired, perhaps a more difficult problem is the organization and control of these sounds into the temporal sound

structures we recognize as rhythm, melody, harmony, and musical form. Although these are musical terms, the same capabilities are needed if we are to encode information into sound in a way that the user of a system can comprehend.

1.2 Levels of Interface

There are at least two levels of interface involved in any computer application. We will refer to these as the *Designer Interface* and the *End-user Interface.* This chapter is concerned with the Designer Interface, sometimes called the *Application Programmer Interface.* We use the former term to emphasize the fact that many of the people involved in the development of interactive applications are not programmers; instead, they program because that is what is needed to create the application.

At this level, the application designer makes decisions that will strongly influence the way the end-user conceptualizes the world. However, the designer's view of the system and its capabilities is defined by the system within which he or she works.

2 Problems

2.1 Lack of Auditory Understanding

Multimedia designers tend to come from a graphical background, and thus most applications are primarily visual. Sound is often used at a trivial level to provide feedback for actions such as button pushes or error conditions. Even when the designer has an understanding of sound in an interactive environment, the tools provided often do not facilitate complex relationships between user, graphics, and sound.

2.2 Few Tools for High-Level Control

Most available tools for sound development are at a low level of abstraction. The situation is analogous to being able to program only at the assembly level – in effect, designers are given too much freedom. Although a low-level system is theoretically flexible, complex applications are prohibitive from a practical standpoint. Faced with this situation, designers may use sound very simply, so that (for example) only commands for triggering auditory events of a long duration would be issued. This will result in a very coarse-grained control, and much of the dynamic, real-time aspect of sound will be lost.

If finer control is exercised, there is the danger of making the structure of the application too complex. The fine-grained timing needed to generate and control complex sound structure can be difficult to achieve in a standard programming language.

One of the most difficult parts in the creation of any interactive interface is the precise control of time. In animation systems, specifying motion paths for a large number of objects can be a daunting task, and several researchers have proposed methods for automated or assisted motion generation [7]. Precise control of time is even more important when dealing with sound. Many simultaneous streams of information can be contained in a sonic environment, and the human auditory system is exquisitely sensitive to minute differences in temporal location of events.

2.3 Control Data Explosion

If users are restricted to low-level models of sound production, a problem arises in the large amount of control data that must be specified to produce interesting results [8]. An example

of this is additive synthesis, a simple and flexible method of sound production. However, this simplicity and flexibility also makes additive synthesis quite difficult to control. This difficulty is not unique to additive synthesis, and is one of the principal reasons that most "synthesized" sounds today are in fact simply playback of digitally sampled sounds.

2.4 Interdisciplinary Teams

As technology becomes more complex, many formerly disparate fields are brought together in interdisciplinary research and development teams. This mode of working has been demonstrated for the past 20 years at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago, and more recently at the Renaissance Experimental Laboratory at the National Center for Supercomputing Applications (NCSA). At these sites and others, scientists, engineers, and artists have collaborated on various projects.

While this interchange of ideas often results in ideas or discoveries that would not otherwise have occurred, some effort can be wasted as team members are forced to learn (often a considerable amount) about each others' code implementation details.

3 What is Needed

3.1 Knowledge Encapsulation

Every system for sound production, no matter how low-level, shapes the user's view of the nature of sound. The importance of this view cannot be overstated. A problem that is intractable with a particular formulation may have a trivial solution if it is stated differently. Experts in the fields of sound synthesis and composition bring their own schemata to bear when creating new techniques.

A system that supports a particular view of sound creation will facilitate the use of algorithms and techniques based on that view. If different sets of knowledge and concepts can be encapsulated into objects or routines in the system, then users can simultaneously make use of multiple paradigms for sound creation.

3.2 High-Level Control

Most designers who are not experts in audio design need high-level functionality, in particular to aid in control of timing, but also in the generation of the large amounts of control data needed to drive many synthesis algorithms. For instance, the designer would be presented with a number of "black box" algorithms, which produced interesting results that could be controlled with a small number of intuitive parameters.

3.3 Low-Level Control

More advanced or adventurous designers would want access to parameters at a low level also. Not only should low-level objects such as oscillators and amplitude envelopes be directly controllable, but the black boxes mentioned above should allow designers to modify low-level parameters of the algorithm.

3.4 Distributed Working Arrangement

An application may be developed through collaboration between several individuals with different fields of expertise. For example, a scientist may create simulation data, which is then visualized by

a graphics programmer or artist, while sound is added by an electronic music composer [9, 10, 11]. A system designed to work in this environment should allow these three individuals to collaborate without being forced to understand each others' code.

4 Auditory Actors

4.1 The Actor Concept

This chapter proposes an organization that satisfies the above requirements. The approach is based on the concept of an *actor*, taken from the field of artificial intelligence (AI) [12]. This concept has also been used in computer music, for example in the interpreted language Act1 [13]. Actors are *active objects* that may have an arbitrarily complex internal state as well as a set of behaviors. Here, the term active implies that a process or thread is associated with each actor. Although this is usually simulated by time-slicing or polling, parallelism is inherent in the concept of actors.

Actors are software objects that encapsulate knowledge and behavior in some way. All communication with and interaction between actors is carried out through message passing. Since each actor hides its internal structure, it is possible to have multiple representations of a single conceptual object. This is important, since in auditory display as in AI, no one representation may be best for all purposes [14].

4.2 Auditory Actors

Auditory actors can respond to messages from the client and interact with each other. As they are active objects, actors will proceed with their behavior if no new messages are received. This approach can be thought of as rule-based, so rather than simply specifying what an auditory actor should do now, a message can specify how the actor should respond to various situations from now on.

All auditory actors have some intrinsic concept of time. This can shift the burden of timing logic from the client to the actor. As an example, consider the case where the client wishes to play a melody. If the timing is handled outside the client application, rhythmic accuracy can be improved, especially if the server is running on a separate computer. However, the client gives up intimate control over when individual events are initiated, so this can be a problem if non-auditory events or processes such as animation are to be synchronized to the melody. The problem of synchronization is addressed below.

Classes of auditory actors are built hierarchically, and can be classified into two general (and not necessarily mutually exclusive) categories. The first category consists of objects that create new waveforms, timbres and types of sounds. These classes of actors could be used to (for instance) simulate bell or trumpet sounds. The second category comprises objects that organize time at a larger scale, such as musical phrases and rhythms. These actors could then send messages to actors from the first category to generate the sounds.

A benefit of message-passing as the only means of communication with actors is that the identity of the message sender is hidden from the recipient. This implies that an actor cannot distinguish a message sent by the client from a message sent by another actor. Building up a network of actors is then much simplified, as the client can test different subsets of the network independently.

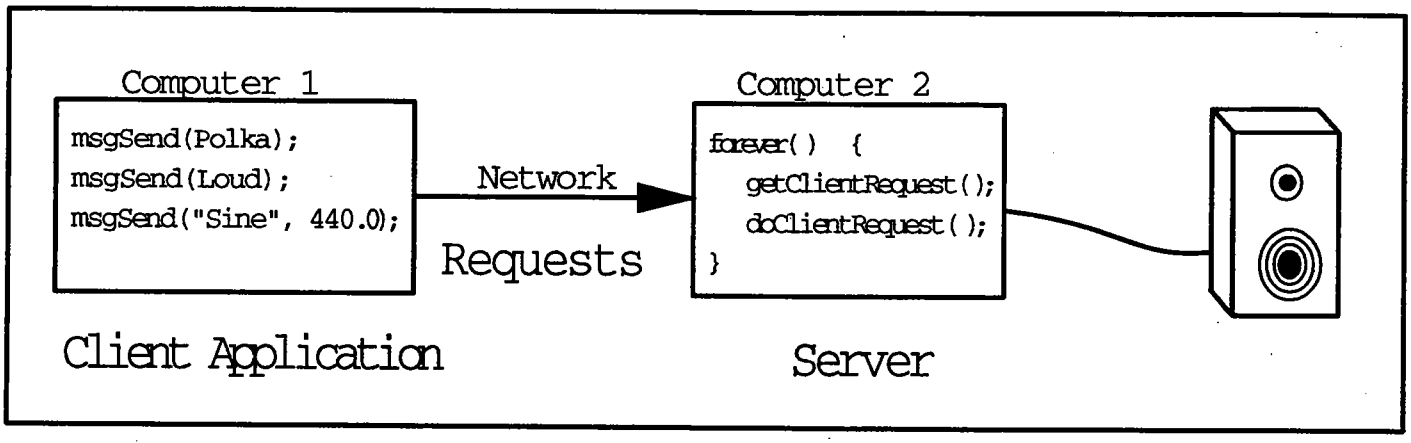

Figure 1: The Client-Server Model.

4.3 How Auditory Actors Can Help

4.3.1 Knowledge Encapsulation

There exist many different and useful methods of generating sounds and the organization of sounds. By consulting with experts in these fields, any synthesis or compositional algorithm can be encapsulated into an auditory actor which would accept messages appropriate to the control of the given algorithm. This actor would then be available for future applications to use. Each of these actors would encapsulate knowledge about a task as well as a way of approaching that task. Unlike libraries of sounds, libraries of actors would be dynamic; that is, although the overall approach to a timbral or compositional task might be determined by the class of actor used, the client would be able to get a wide range of behaviors and effects from a single class.

4.3.2 High-Level Control

Once a respectably sized library of actors has been developed, the high-level control needs of most applications can be served by new configurations and combinations of existing actors. If an application has needs that are too radical or undefined to use existing high-level actor controls, lower-level messages can be sent to existing classes of actors until the desired functionality is clear. At that point, a new actor class can be derived, and the low-level logic can be moved from the client to the new actor.

4.3.3 Low-Level Control

Auditory actors can also provide access to arbitrarily low-level parameters of algorithms. Since actors have access to all parameters of the synthesis algorithms they control, messages can be provided that simply pass parameters from the client through the actor to the underlying synthesis algorithm.

4.3.4 Distributed Working Arrangement

We will show how the use of actors allows a generic designer interface to be created, which among other things, will allow the sound portion of an application to be modified without modifying the graphical or simulation code.

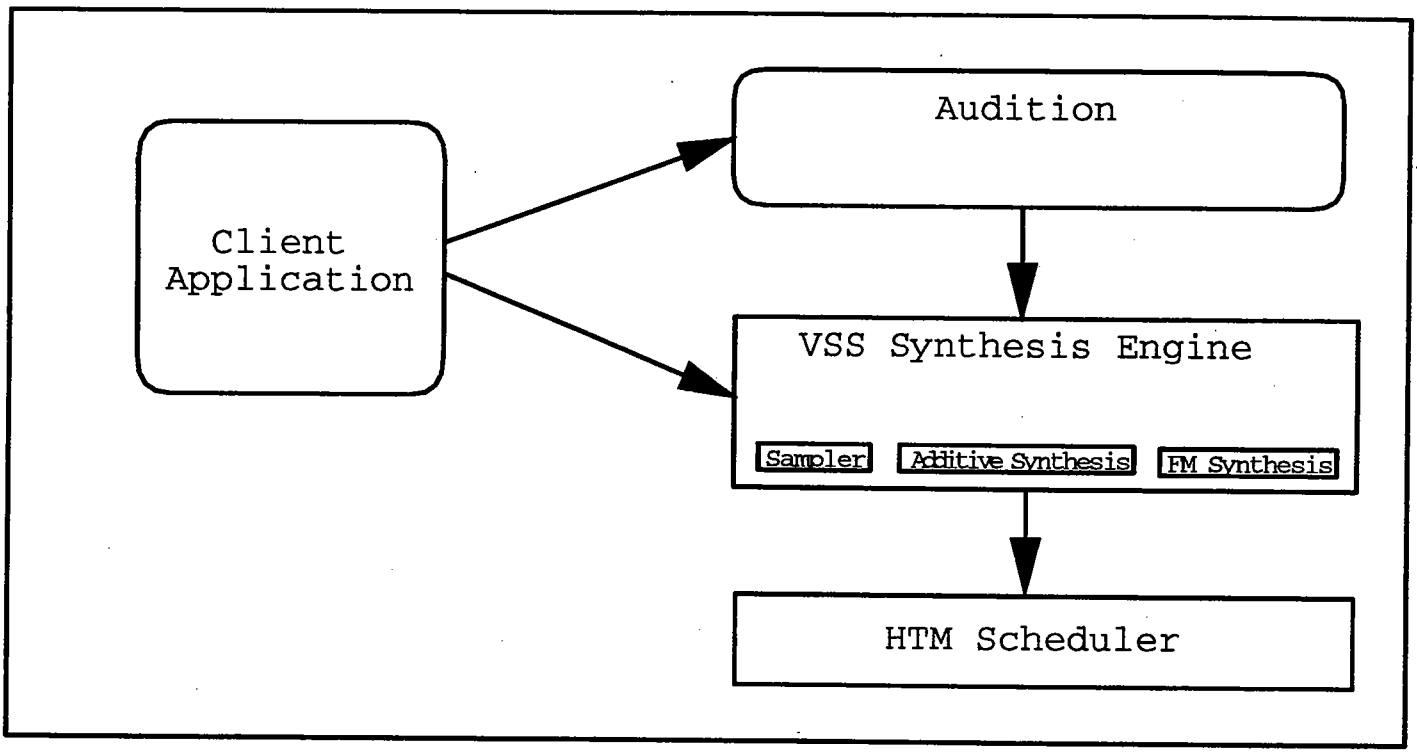

Figure 2: System organization.

5 Implementation

5.1 Overall System Organization

This section describes the implementation of a system that incorporates the ideas described above. This system is built on the *client-server* model. In this model, the application program which needs sound is called the *client*. The client sends requests to the *server*, which is another program, usually running on a different computer. The server then fulfills the client's requests to the best of its ability (Figure 1).

The system of auditory actors is called **Audition**. Clients can send messages to **Audition**, which is built on top of an underlying synthesis engine, **VSS**.[1] **Audition** handles handles issues of timing, parameter mapping, and logic, and then sends lower-level messages to **VSS** to generate sound. **VSS** in turn is built on top of a scheduler called **HTM**.[2] This organization is shown in Figure 2. The server runs on Silicon Graphics Indy or Indigo series workstations.

Audition is implemented in C++, which is a strongly-typed compiled language. As such, it lacks some features of interpreted object-oriented languages like Smalltalk. Some of these features, such as run-time checking of argument syntax and method resolvability have been simulated in **Audition**. This gives the client many of the advantages of an interpreted language. Since message-processing is a relatively infrequent and inexpensive task compared to the bulk of the server's duties, we retain most of the efficiency of the underlying compiled language.

One advantage of the client-server model is that the client program need not be linked with a large, complex system. Instead, only the message-passing routines have to exist in the client application. At present, clients can send messages to **Audition** from C or C++ modules.

[1]VSS is a sound server under development at the National Center for Supercomputing Applications.
[2]HTM was developed by Adrian Freed at the Center for New Music and Audio Technologies, UC Berkeley.

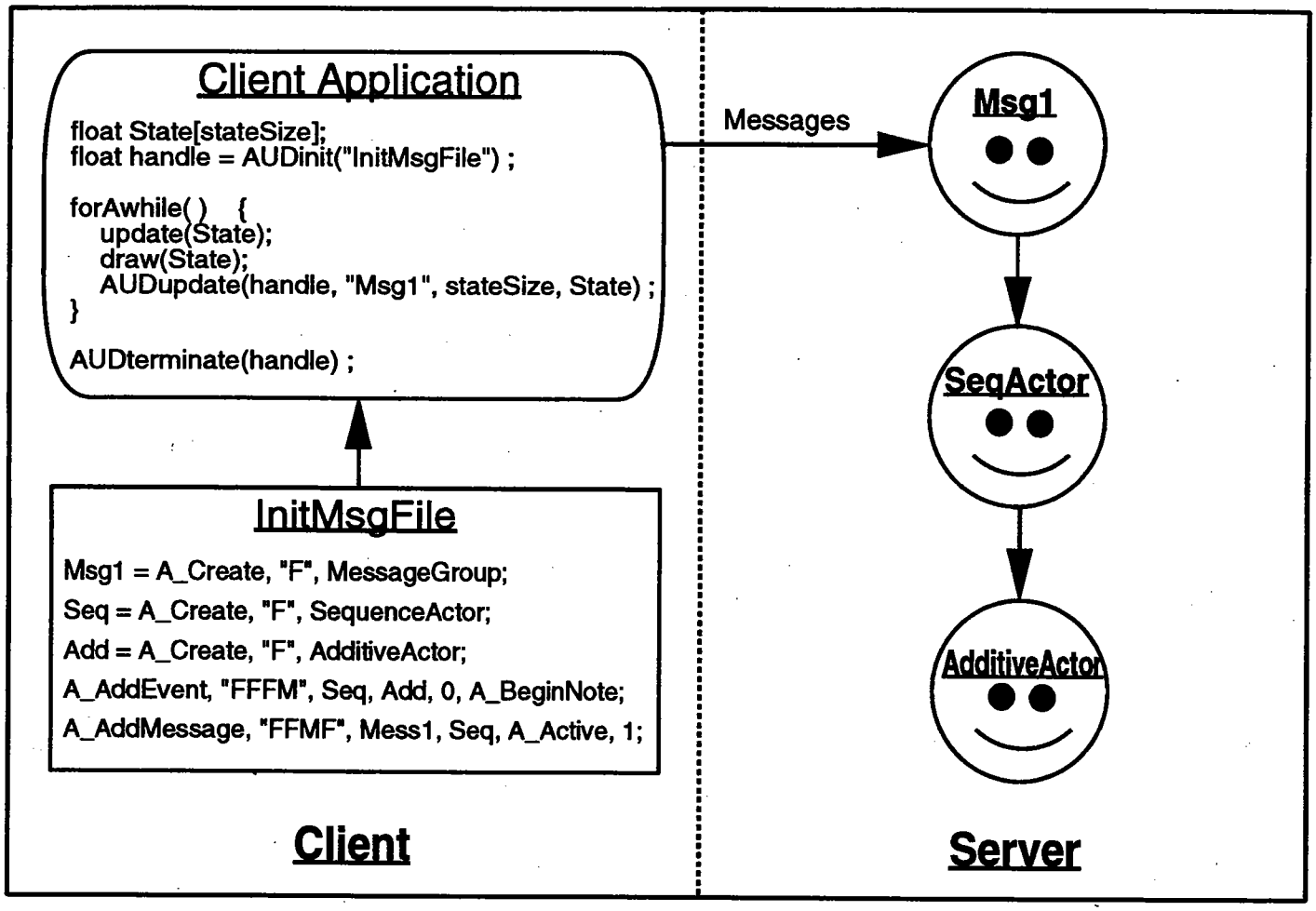

Figure 3: Generic application programmer interface.

There are two ways a client can communicate with **Audition**. For designers who do not wish to delve into the intricacies of sound design or the **Audition** system, a programmer's interface is provided that allows the client to set up and use actors while knowing very little about the server or its capabilities.

Alternatively, the lower level message passing routines may be called directly for finer control. Details of the organization, actor classes, and message syntax are given in [15].

5.2 Client Organization

5.2.1 Generic Application Interface

In order to use this interface, there are two things that must be done. First, the client must tell the server which class of actor it wants to use and how it wants that actor configured. This configuration can include information such as the number of parameters that will be sent with each update message and how to interpret these parameters. Then, the client sends data to the actor, either at regular intervals or whenever a state in the application changes. The data is in a generic form, and consists of an array of floating-point numbers. Figure 3 shows how this is organized.

Setting up and configuring the actors is done through the use of an initialization file, called

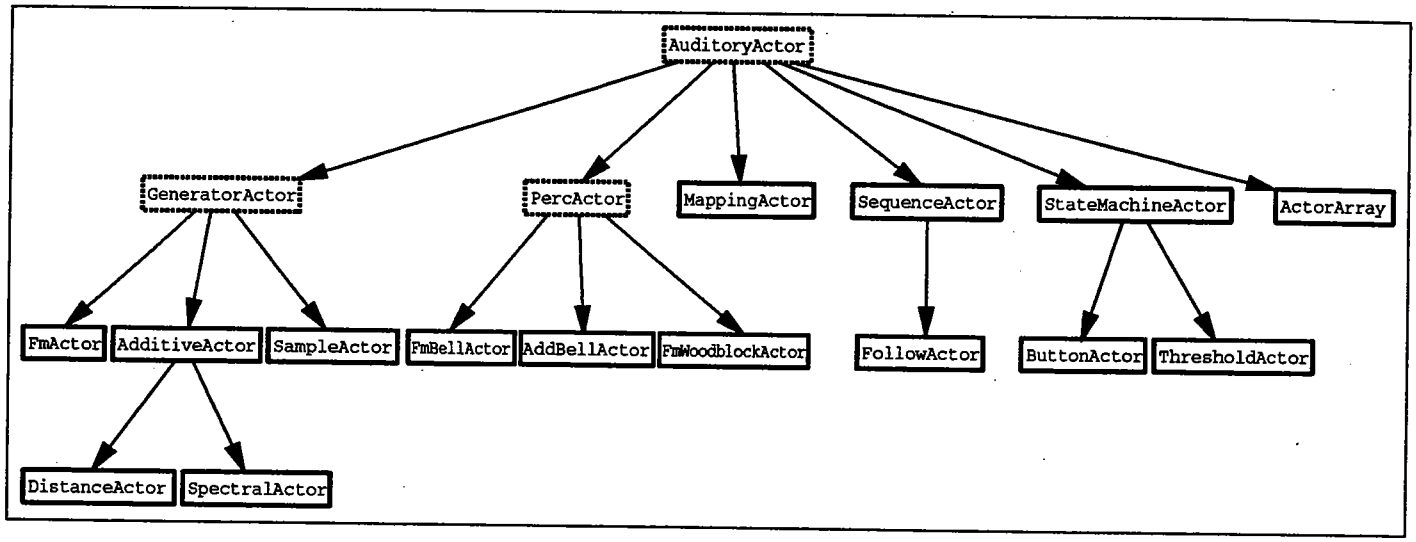

Figure 4: Auditory actor derivation hierarchy.

`InitMsgFile` in the Figure. This file contains messages to be sent to the server. If the designer understands enough about the message format, they can modify this file to change the behavior of the actors. Otherwise, a number of these files are provided with the system to set up the **Audition** system in different ways.

The initialization file will create an actor for the client to use and return the actor's handle. This handle will then be used to send messages to the actor. As can be seen in Figure 3, the client's actor may create a network of other actors to use in making sound, but the client will only communicate directly with the one actor at the top of the Figure. In fact, it need send only one type of message to this actor, which updates the actor as to the state of the client (`ABSupdate()` in the Figure). The actor labeled `appActor` in the figure serves mainly to interpret the incoming data, and for more complex applications it may be custom designed.

5.2.2 Explicit Application Interface

Application designers who require more control can explicitly send messages to the **Audition** system. Messages are sent by calling message-passing functions which can take a variable number of arguments of a few simple types (float, character string, float array). No argument checking is done at compile time, instead each auditory actor dynamically checks the message syntax and decides if it can fulfill the request. Messages are provided to create, modify, debug, and delete actors, as well as messages to aid in debugging client communication with the server.

5.3 Server Organization

5.3.1 Actor Hierarchy

The derivation hierarchy of the Actor classes is shown in Figure 4. The arrows in this figure denote "is a" relationships, so a `SampleActor` "is a" `GeneratorActor` which "is a" `AuditoryActor`, in the same way that a dog "is a" mammal which "is a" animal. The implication here is that as we move down a path in the hierarchy, each actor class inherits the characteristics of the class it is derived from; and then specializes or adds features.

The classes surrounded by dotted lines are *virtual classes*, which means that they are just there

to conceptually group together classes derived from them, and the client cannot create instances of these classes. `AudEvents` are used by `SequenceActors` to create sequences of events, and at this time, the client is not given direct access to them. The client can request **Audition** to create an instance of any other class of auditory actor and return a handle. We briefly describe the existing classes of actors.

Generator Actors Subclasses of `GeneratorActor` are the only auditory actors that send messages directly to the synthesis engine to create sound. Each instance of `GeneratorActor` can be thought of as a synthesizer, in that it can simultaneously play multiple notes. Messages can be sent to the actor or to any note currently being sounded by the actor.

`FmActor`, `AdditiveActor`, and `SampleActor` communicate directly with the corresponding algorithms in **VSS**. They allow the client (and other auditory actors) access to all parameters of the underlying synthesis methods, so that low-level control is not lost. The `GeneratorActor` class does some parameter remapping, such as allowing the client to set frequency parameters by using either Hertz (cycles per second) or pitch (a logarithmic measure with reference to a chosen frequency).

`GeneratorActors` also generalize certain messages, allowing the client to set parameters which may not actually exist in the algorithm. For instance, once the frequencies for all partials (sinusoidal components) have been specified for a note generated through additive synthesis, the client may wish to change the frequency of the note. However, a note generated through additive synthesis does not have a single overall frequency, so ordinarily all partials would have to be re-specified in order to change the frequency of the note. An intuitive interpretation of the message is made by assuming that the frequency of the entire note is same as that of the partial with the lowest frequency. With this interpretation, all the partials' frequencies are then shifted logarithmically so the resulting frequency of the lowest partial is the one specified by the client's message.

Bell Actors These are actors that implement well-known "recipes" for simulating bell sounds. `FmBellActor` uses frequency modulation and an empirical formula by Chowning [16]. `AddBellActor` is based on an additive synthesis model by Risset, created from analyses of real bell sounds [17]. For each bell tone, the client can set the pitch, amplitude, and duration independently. A more intuitive set of parameters might include size, wall thickness, and material composition.

Other models of this type are planned, so that users can easily choose from a library of synthesis templates. Sampled (recorded) sounds can also be used, but the advantage of parametric models is that a single model can be tailored to fit a user's needs simply by setting parameters.

Sequence Actors Many applications need to initiate a series of events whose relative positioning in time is known in advance. If the events consist either of notes or control information (pitch change, modulation, etc.), the resulting construct is the familiar MIDI sequencer. The `SequenceActor` is a generalization of this idea.

A `SequenceActor` is a list of time-stamped events, which are messages sent to actors (it can even send messages to itself, for instance to change tempo or add events). Any message that the client can send to an actor can be added to a `SequenceActor` and then played back later. With the addition of a few simple classes of actors that generate boolean or numerical responses to input, `SequenceActors` can also be used to conditionally send messages.

A problem alluded to earlier was the loss of control when a sequence of events is initiated on the server. In an interactive graphical application, the frame update rate may vary unpredictably,

which can cause the graphics to drift out of synchronization with the sound. There are two possible solutions to this: either the visuals must be forced to match the timing of the sound, or the sound processes must change tempo to match the visuals.

Since auditory processes are much more sensitive than visuals to tempo changes, it would appear that the proper approach is to force the graphics to follow the timing constraints imposed by the sound. This presents implementation difficulties, and would force an unconventional structuring on the client application. For the present, the `FollowActor` class allows the client to send metronome messages at each beat, and adjusts its tempo accordingly. As long as the graphical processes do not distort timing to the point that the rhythmic effects of the sound are lost, this should suffice for most purposes.

5.3.2 Synthesis Engine

The synthesis algorithms currently implemented are additive and FM synthesis, and sample playback.[3] At the heart of any sound generation system, the scheduler makes decisions regarding the relative priorities of processing control signals versus audio sample generation. The current system is implemented on top of **HTM**. This scheduler provides for fast communication between client and server, as well as taking care of scheduling CPU time for time-sensitive tasks such as generating samples and servicing client requests. Communication between client and server is achieved via udp, a fast Ethernet protocol with no error checking. The server and client can reside either on separate machines or on the same machine.

6 Test Applications

The system was tested by distributing the libraries and documentation to the developers of a number applications which were shown at VROOM, an exhibit of virtual reality applications at SIGGRAPH '94. All of these applications were developed for the CAVE, a projection-based virtual reality system being developed at the Electronic Visualization Laboratory [18]. THe scope of the applications ranged from

In all, 39 applications from various disciplines were developed for VROOM. Most of these dealt with scientific visualizations of data. The time constraints were quite severe, and constituted a good test of the development time needed by naive users to create an interactive sonic environment. All development for applications had to be completed within approximately 4 months of approval, with even less time available for the audio development. Since these were interactive applications to be experienced by the general public, they had to be robust, which added to the difficulty of these constraints.

At SIGGRAPH 1992, Showcase 92 also demonstrated (on a smaller scale) several VR applications, using a sound library that provided a few standard MIDI functions, such as note on, note off, pitch bend, etc. Briefly, the Showcase 92 system allowed application developers to control MIDI-compatible synthesizers.

A brief description of three representative applications is now given. A more complete description of the applications can be found elsewhere [19, 20].

[3]The synthesis algorithms are being developed by Kelly Fitz and Camille Goudsene (NCSA, UIUC), and the first author.

6.1 Evolution of Shape and Sound

This application demonstrates the use of genetic algorithms to evolve shapes and corresponding musical accompaniment [21]. It was originally shown at Supercomputing '93 using an early implementation of some of the organizational ideas presented here. Users evolve shapes and musical "styles" by picking their favorites out of four that are presented at a time. Users hear the music associated with the currently picked object.

The sound server is sent a small number of parameters each time a new shape is selected. These parameters are the surface area, "smoothness," and other such parameters of the shape. The interface object on the server is called an `Orchestra`. An `Orchestra` object communicates with a number of `Player` objects. Each player contains a number of `Phrase` objects (short segments of precomposed musical data), and can play variations on them in different orders, depending on the input. The goal is to parametrically control the stylistic attributes of music in real-time.

This application was reimplemented in the **Audition** system, and several improvements will be made to the rule sets. The original implementation generated its audio output through MIDI, and this was replaced by real-time synthesis.

6.2 Alpha Shapes

An alpha shape is a generalization of a convex hull, which is a two-dimensional "skin" wrapped around a cloud of points. This application shows alpha shapes of various sets of points, and uses audio as well as visuals to depict the propagation of a wave through the shape.

Different features of the wave and its history are mapped to different sound parameters. The smoothness of the wave and the topological connectivity of the shape are reflected in the amplitude envelopes of additive synthesis sounds, and the combinatorial size is reflected in the frequency of the partials.[4]

6.3 Artificial Life

Artificial life (AL) is a field of growing popularity whose purpose is (usually) to simulate behavior. In AL, the designer specifies low-level rules, and the high-level behaviors that result are implicit in the specification. This aspect of AL is termed *emergent behavior*. In this application, users can observe and interact with several species of artificial organisms, called `Flyers`, `Skaters`, `Aggressors`, and `Stills` (plant-like creatures).

The overall state of the environment will be reflected in abstract rhythms and timbres generated by a `RhythmActor` (still under development). Individual creatures will also generate sounds when engaging in certain behaviors, such as attacking or feeding[5].

7 Conclusions and Future Work

The concept of an auditory actor constitutes an organizing principle for the creation of sound in interactive computer applications. These constructs allow arbitrarily high-level control of sound structures, while also allowing access to low-level parameters of algorithms.

[4]The Alpha shapes application is being developed by Ping Fu, Gilles Bourhis, and Robin Bargar (UIUC, NCSA), Herbert Edelsbrunner and Ulrike Axen (UIUC, Computer Science), and Insook Choi (UIUC, Music and Computer Science).

[5]The Artificial life application is being developed by Terry Franguiadakis and Michael Papka (EVL, UIC), and the first author.

The abstraction provided by actors residing on a remote server allows parallel, independent development between the various parts of the application. In this way, experts in different fields can collaborate on an application while only needing to understand a limited subset of the whole application. In addition, improvements and additions to the system of actors need not affect the client programs.

Several other actors are being developed or designed. Among them are generator actors for granular synthesis and MIDI. To create patterns, a set of "decision makers" will be created that will return numerical values based on input and internal state. Examples of these are random number generators, boolean and mathematical operations, Markov chains, and finite state automata.

Actors are being designed to create commonly used sounds such as buzzers and musical instruments, as well as actors to abstract the functions of devices such as a multi-button mouse or the wand (a six-degree of freedom pointing device used in the CAVE).

One area that deserves exploration is being able to synchronize either the client or the server to each other. Since the server is designed to simply and precisely control time, timing functions for animations could possibly be handled by an actor. Many animations take their timing cues from the soundtrack, so this would be a familiar mode of working for animators.

8 Acknowledgments

This material is based upon the work supported by National Science Foundation under Grant No. CDA-9303433, which includes support from the Advanced Research Projects Agency. The authors thank Ulrike Axen, Insook Choi, Robin Bargar, Camille Goudsene, and Kelly Fitz for ideas, hacking, and testing. Thanks also to Maxine Brown, Gary Lindahl, Dana Plepys, and Maggie Rawlings for all their help, and to all the other members of the Electronic Visualization Laboratory.

References

[1] Mathews, M. V. "An Acoustical Compiler for Music and Psychological Stimuli." *Bell system Tech. J.* **40** (1961).

[2] Mathews, M. V. "The Digital Computer as a Musical Instrument." *Science* **142** (1963).

[3] Loy, G. "Composing with Computers—Aa Survey of Some Compositional Formalisms and Music Programming Languages." In *Current Directions in Computer Music Research*, Ch. 21, 291–396. Cambridge, MA: MIT Press, 1994.

[4] "MIDI 1.0 Specification." International MIDI Association, North Hollywood, 1983.

[5] Puckette, M. "Interprocess Communication and Timing in Real-Time Computer Music Performance." In *Proceedings of the ICMC.* San Francisco: CMA, 1986.

[6] Scaletti, C. "The Kyma/Platypus Computer Music Workstation." *Comp. Music J.* **13(2)** (1989).

[7] Reynolds, C. W. "Flocks, Herds and Schools: A Distributed Behavioral Model." *SIGGRAPH 87* (1987): 25–34.

[8] Wawrzynek, J. "VSLI Models for Sound Synthesis." In *Current Directions in Computer Music Research*, Ch. 10, 113–148. Cambridge, MA: MIT Press, 1994.

[9] Cruz-Neira, C., J. Leigh, C. Barnes, S. Cohen, S. Das, R. Englemann, R. Hudson, M. Papka, T. Roy, L. Seigel, C. Vasilakis, T. DeFanti, and D. Sandin. "Scientists in Wonderland: A report on Visualization application on the CAVE Virtual Reality Environment." *IEEE Symposium on Research Frontiers in VR*, 1993.

[10] Bargar, R., and S. Das. "Virtual Sound Composition for the CAVE." Proceedings of the 1993 International Computer Music Conference, Tokyo, Japan, September, 1993.

[11] Mayer-Kress, G., R. Bargar, and I. Choi. "Musical Structures in Data From Chaotic Attractors." Technical Report CCSR-92-14, Center for Complex Systems Research, The Beckman Institute, University of Illinois at Urbana-Champaign. Santa Fe Institute International Conference on Auditory Display, Santa Fe, NM, October, 1992.

[12] Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages." *Art. Int.* **8(3)** (1977): 323–364.

[13] Lieberman, H. "Machine Tongues IX: Object-Oriented Programming." *Comp. Mus. J.* **6(3)** (1982): 8–21.

[14] Roads, C. "Interview with Marvin Minsky." *Comp. Mus. J.* **4(3)** (1980): 25–39.

[15] Das, S. "The Audition Sound Server." Electronic Visualization Laboratory, University of Illinois at Chicago, 1994.

[16] Chowning, J. M. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation." *J. Audio Eng. Soc.* **21** (1973): 526–534.

[17] Risset, J.-C. "An Introductory Catalogue of Computer Synthesized Sounds." Bell Telephone Laboratories unpublished memorandum, Murray Hill, New Jersey, 1969.

[18] Carolina Cruz-Neira and Daniel J. Sandin and Thomas A. DeFanti. "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE." *SIGGRAPH 93 Proceedings*, Anaheim, California, (1993): 135–142.

[19] Das, S. "An Organization for Interactive Programmatic Control of Computer-Generated Sound." Electronic Visualization Laboratory, University of Illinois at Chicago, University of Illinois at Chicago, Electrical Engineering and Computer Science, 1994.

[20] Petrovich, L., K. Tanaka, D. Morse, N. Ingle, J. Ford Morie, C. Stapleton, M. Brown. "VROOM." In *Visual Proceedings, SIGGRAPH 94*. New York: ACM SIGGRAPH **6** (1994): 218–267.

[21] Das, S., T. Franguiadakis, M. Papka, T. A. DeFanti, and D. J. Sandin. "A Genetic Programming Application in Virtual Reality." *IEEE Proceedings of Computational Intelligence Conference on Evolutionary Computation*, Orlando, FL, 1994.