Aura II: Making Real-Time Systems Safe for Music

Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213 USA
+1-412-268-3827
rbd@cs.cmu.edu

ABSTRACT

Real-time interactive software can be difficult to construct and debug. Aura is a software platform to facilitate highly interactive systems that combine audio signal processing, sophisticated control, sensors, computer animation, video processing, and graphical user interfaces. Moreover, Aura is open-ended, allowing diverse software components to be interconnected in a real-time framework. A recent assessment of Aura has motivated a redesign of the communication system to support remote procedure call. In addition, the audio signal processing framework has been altered to reduce programming errors. The motivation behind these changes is discussed, and measurements of run-time performance offer some general insights for system designers.

1. INTRODUCTION

Aura is a software framework for building real-time interactive systems. [12] It has evolved and grown for almost 10 years, and has roots in the CMU MIDI Toolkit [5, 9], created almost 20 years ago. Building on experience gained from using Aura and related systems, Aura II is my latest attempt to provide a solid foundation for interactive systems. Aura II embraces a number of design principles I have found to be effective. The goal of this paper is to describe the design philosophy and experience that has led to Aura II and to discuss some lessons learned. I will also include some performance measurements that may help others to make intelligent design choices.

In the next section, I present the main concepts that form the basis of Aura. Section 4 provides an example for further motivation. In Section 4, I describe in greater formality the approach to scheduling, and Section 5 discusses various aspects of message passing in which many systems issues arise. Section 6 describes the structure of the audio processing system, and this is followed by a summary and conclusions.

2. ORIGINS OF AURA

Aura was designed to replace the CMU MIDI Toolkit (CMT), an elegant system to help students write interactive MIDI applications with a minimal knowledge of programming. CMT adds essentially one thing to the C programming language: a function, `cause`, that calls a function at some time in the future. E.g., one can write

```
cause(0.3, midi_note, chan, pitch, 0)
```

to send a note-off message (velocity = 0) after 0.3 seconds. While simple in appearance, `cause` can be used to run multiple concurrent processes. Each process is implemented as a procedure that performs an action and then "cause's" itself to resume execution in the future. All procedures run to completion, eliminating almost any concern over shared variables, reentrant code, critical sections, and preemption. (And although CMT "processes" are not true processes that can suspend and resume at any point, they handle most event-driven programming tasks quite well.)

In the early 90's it became clear that signal processing was going to move from special purpose VLSI and DSP chips to general purpose processors. [8] At the same time, real-time video and computer graphics was becoming an attractive new performance medium. While computer graphics tasks could take tens of milliseconds, MIDI and control processing must be much more responsive, and audio processing adds a hard real-time processing requirement: no task should ever hold up computing and cause a buffer overflow or underflow. While the simple non-preemptive model of CMT is successful in its domain, it cannot handle the wide range of processing tasks and latencies posed by general interactive multimedia.

Also in the 90's, I worked briefly on constraint-based graphical user interface systems [16], and the idea of connecting various objects with constraints offered an interesting model for rapid prototyping and programming by novices. "Connect the position of the slider to the frequency of the oscillator" seems much simpler than "subclass the slider object and add a method that calls the `set_hz` method of the oscillator object." The idea of connecting attributes of objects together is something like patching an analog synthesizer or wiring a network of MIDI synthesizers, which seems quite intuitive. This same idea works successfully in MAX [20] and related software.

Thus, Aura I (originally named "W") [12] incorporates three main ideas: (1) Where possible, use non-preemptive scheduling (as in CMT) and `cause` to support many tasks, (2) Use "real" preemptive threads to allow time-critical processing to preempt longer-running computations, and (3) Use objects with attribute/value interfaces, and use constraint-like connections to create applications. In retrospect, (1) and (2) seem to be very good ideas, whereas (3) does not. This and other problems have motivated Aura II, which is discussed mainly in Sections 5 and 6. In what follows, I will use *Aura I* to refer to Aura using the attribute/value object model, and *Aura II* to refer to the current version based on remote procedure call. I will use *Aura* when comments apply to both versions.

3. AN EXAMPLE APPLICATION: THE WATERCOURSE WAY

To illustrate how Aura is used and why current systems are either unsuitable or only partial solutions, I will describe a recent work, "The Watercourse Way" [10], an interactive piece for chamber orchestra, dancer, and electronics. The main technical focus of

this work is a sound synthesizer controlled by water movement. The water is sensed by reflecting light onto a screen, creating beautiful moving images. The reflected light is captured by a video camera, digitized, and further processed to derive time-varying spectra for three synthesized tones. [11] In addition, the video is texture-mapped onto the surfaces of an animation that is projected in the concert space. Four channels of audio input capture various instrumental combinations and four output channels drive a quadraphonic sound system. Figure 1 illustrates the entire system, and Figure 2 shows an audience view of the dancer and images.

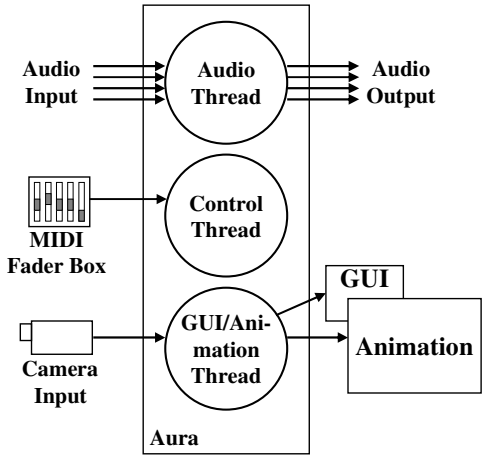

Figure 1. System configuration for The Watercourse Way.

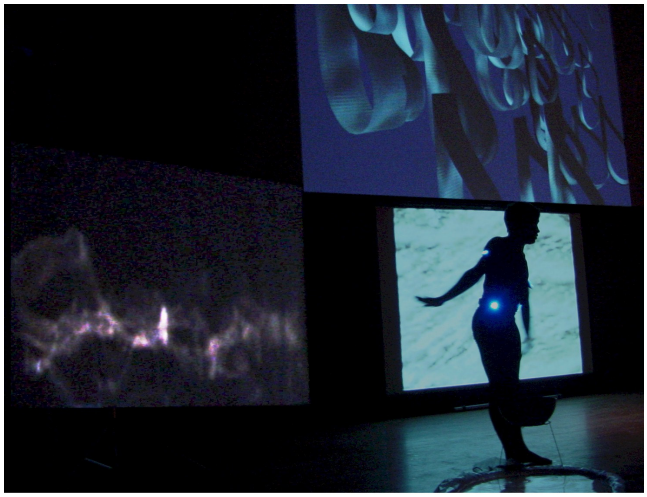

Figure 2. Performance of "The Watercourse Way." Light (left) reflected from water (bottom right) is used to synthesize sound and is also texture-mapped onto animation (upper right). The sound and image are modulated when the dancer touches the water. Live musicians are not shown here.

In this system, low-latency audio is required to process live instrument sounds. At the same time, the camera input and processing runs in lock-step with the 30-frame-per-second video input. Clearly, the audio thread must run in a separate thread to avoid blocking for up to 1/30 of a second waiting for the next video frame. The animation involves a physical simulation of a stiff material flexing under applied forces. This, and the OpenGL rendering of the model, cannot be easily expressed by combining pre-fabricated image operations, so it is important that Aura is fully programmable, not only for audio rendering, but for control, animation, and graphical user interfaces. While other systems offer very powerful interfaces for signal processing or even video processing, few systems have the generality, programmability, and extensibility offered by Aura.

In "The Watercourse Way," as in all Aura applications, objects are segregated into different threads, or *zones*, such that high-latency objects share one thread and low-latency, time-critical threads share another. This approach is described in more detail in the next section, and the message passing system used for communication between objects is described in Section 5.

4. LATENCY MONOTONIC SCHEDULING

Real-time systems for interactive performance must deliver high performance and at the same time be easy to create and modify. Musicians and composers simply do not have the resources of the Mars Rover team to build and debug complex real-time systems. A traditional approach to real-time systems is to implement each task with its own, schedulable thread, and use synchronization primitives (locks and semaphores) to guard access to shared variables through which tasks communicate. [1] The problem with this approach is that it is easy to make mistakes and hard to find them. Forgetting to lock a data structure can cause intermittent errors that cannot be reproduced easily and may disappear when debugging code is added. The traditional approach also adds complexity to the code and is usually best approach with a careful system-wide design process that is not conducive to rapid prototyping.

An alternative approach is to use non-preemptive threads as in CMT. Here, a task runs to completion or to some point at which the code explicitly releases the processor. Since no unexpected preemption can take place, no shared variables need to be locked or unlocked. However, as mentioned earlier, this approach breaks down when there is a mix of long-running computations and low-latency deadlines.

Now imagine running a CMT-like non-preemptive program on each of several independent computers, where one computer computes audio, another manages a graphical user interface and renders animation, and yet another handles control and MIDI data. If tasks are segregated according to their latency requirements, non-preemptive scheduling can be effective. In fact, most music processing is time driven, so CMT-like systems execute tasks/procedures according to the time at which they are scheduled. Assuming that all deadlines can be met, this form of scheduling, "earliest deadline first," is known to be optimal.

Aura can run in this fashion on multiple computers, but the typical system shares one processor among three threads that handle graphics, control, and audio, respectively. Each thread is managed by an Aura *zone*, which also provides inter-zone communication, memory management, and scheduling. Zones use fixed priority scheduling where priority decreases monotonically with the zone's *latency* requirement. Here, *latency*, is the maximum amount by which a task can fall behind real time. For audio computation, latency is the time it would take for the audio buffer to underflow. For animation, latency is roughly the frame period.

Using these concepts, we can make some precise statements about the real-time performance of Aura. For the sake of brevity, I will

assume that each zone has a maximum latency requirement L_i and that each latency requirement L_i is an integer multiple of the next-higher priority zone's latency requirement L_{i-1}. (For example, requirements could be 20, 2, and 1 msec for graphics, control, and audio zones, respectively.) I also assume that in the worst case, all tasks in a zone are scheduled to run simultaneously, but that no task is rescheduled in less time than the zone's latency. C_i is the total worst-case total execution time of all tasks in zone i:

$$C_i = \sum_j d_{ij},$$

where is d_{ij} is the worst-case execution time of task j in zone i.

The latency experienced by a zone is the execution time of the tasks in the zone (C_i) plus the execution time of tasks in higher priority zones which might preempt this one. Note that zone j can preempt zone i exactly L_i/L_j times if $L_j < L_i$. Therefore the constraints to guarantee that all latency requirements are met are:

$$L_i > \sum_{j=1}^{i} \frac{L_i}{L_j} C_j, \text{ for } i > 1$$

In actual practice, computation may not be so periodic and predictable, so more elaborate analysis may be required to establish tighter bounds. Alternatively, performance can be measured experimentally. The Aura scheduler can measure the actual latency of each zone and also detect when an object fails to meet the latency requirement. In summary, tasks are assigned to zones according to their latency requirements. Typically, only a few zones are needed, so most computation can be run in a simple, non-preemptive fashion.

5. MESSAGE PASSING

When objects are segregated into zones, and shared memory is not allowed, how do objects communicate? In Aura, communication is by message passing. Messages are delivered by either setting some data within an object or by invoking an object method. In the case of method invocation, the method runs to completion and returns, after which the next message is delivered. Thus, message delivery is performed synchronously with respect to other computation within the zone, and no locking or synchronization is required.

Some explanation is now in order pertaining to shared memory in Aura. The preceding discussion has made it clear that zones operate as if they exist on separate computers in separate address spaces. In the implementation, zones actually share a single address space, but each zone has its own set of memory locations and never accesses the memory of another zone with the exception of some FIFO communication buffers. For each pair of zones, there is a single-writer, single-reader FIFO buffer through which messages can be passed. The advantage of this arrangement is that these FIFOs can be implemented safely without locks, which would require an operating system call per FIFO operation. As measurements will show below, this offers a very efficient mechanism for communication. In fact, it is faster to send a message in this manner than to lock and unlock a shared variable.

The Aura approach to low-latency computation can be viewed as a generalization of a common implementation practice. Often, programmers will devise a standard graphical user interface and rely on either timer callbacks or audio system callbacks to perform real-time processing. It is common to use FIFOs for communication between the main program and the callback. For example, this technique is described by Zicarelli [24] and is also used in MAX. Aura generalizes this approach by viewing both the "user interface" and the "callback thread" as instances of the same thing: a zone in which objects exist and execute. The model is generalized further by allowing more than two zones.

5.1 Attribute/Value vs Procedure Call

To support this generalized view of real-time computation, Aura I introduced a special object model in which objects expose settable attributes and received messages of the form "set *attribute* to *value*." Unlike traditional object-oriented programming "messages"—really a form of procedure call with parameters pushed on the run-time stack—Aura messages are "real" data that can be copied to another zone or address space for remote delivery.

While the motivation for Aura I's attribute/value interface was convincing at the time, experience has led me to believe that this is not a particularly good idea. There are many cases where one wants to invoke an operation with multiple parameters in a single operation, e.g. "connect input *a* to channel *i* of output *b.*" In Aura, this example requires two messages: "set `channel` to *i*" followed closely by "set `a` to *b.*" Splitting what is logically a single operation into a protocol consisting of multiple messages is not good programming practice. (Originally, Aura supported sequences of attribute/value pairs, but it was awkward to construct these messages and they fell into disuse.)

A second observation is that in practice, many messages are directed to specific objects. While the original concept was that messages would flow out of objects, over connections, and into other objects (the behavior resulting from an Aura "send" operation), Aura also provides a "send_to" operation where the destination object is provided explicitly. This "connection free" style of messaging is often preferred because the programmer can direct different messages to different objects. Also, this style tends to be more readable. Since connections are often set up by one part of the program and messages are sent in another, it is difficult to keep a mental image of what objects are connected where. On the other hand, the "send_to" style always names the receiver at the point where the message is sent, making the program easier to understand. This is in contrast to visual languages like MAX, where the connections are part of the program "syntax" and therefore are easier to follow. This is more like OSC, where the message sender explicitly specifies a destination.

The biggest change in Aura II, therefore, is the replacement of attribute/value messages with remote procedure calls. Aura II uses the same low-level message transport mechanism as Aura, but the code to form messages and receive them has been completely replaced to support passing multiple parameters to methods. This work provides an interesting opportunity to compare the performance of the two styles of messages.

5.2 Type Checking and Naming

Remote access, whether through an attribute/value mechanism or a remote procedure is much easier to debug when types are checked automatically. Programmers need help to guarantee that when, say, a 32-bit integer value is sent from object A, the bits will be interpreted as an integer when they arrive at object B. Types can sometimes be checked at compile time, but with

flexible message systems, types must often be checked at run time. Typically, the message includes some specification of the data types as well as the data, and the receiver checks that the specification is for the expected types (the receiver can also perform different actions according to the type).

In Aura I, messages do not carry explicit type information. Instead, types are associated with attributes themselves. In practice, to send an integer value, the sender calls

`set_integer(`*`attribute`*`, `*`value`*`)`

and the `set_integer` method checks that *`attribute`* has type *integer*. Since attributes are global, attribute name conflicts can to arise. One object might declare `maximum` to be an integer, and another will try to make it a float. This problem was solved using the somewhat ugly but effective practice of appending a type character to the name, so `maximumi` is an integer attribute, and `maximumd` is a (double-precision) float attribute.

In Aura II, the attribute is replaced by a method name. Internally, the system appends type information to the name to form what is called the signature. On the sending side, a function is generated to send each message, so ordinary compile-time type checking insures that the message is well-formed and that the signature matches the actual parameter data. On the receiving end, the signature is used to locate a message handler. If a handler is found, then it is known that the parameters have the correct type for the receiving method in the receiving object. So far, no special naming scheme is in use, so users could be confused when a message is not received due to a type mismatch. The plan is to create two classes of messages. The *standard* message *must* be handled by the receiver, otherwise an error is reported. The *optional* message is intended for broadcasting data to objects that may only be interested in certain messages. These messages can be dropped without raising errors.

5.3 Performance Measurements

Benchmarks were used to measure the time Aura I and II take to send messages. Further analysis using execution profiling revealed some general information about where time is spent. Since sending parameter updates is an important part of any interactive system, designers can benefit from a better understanding of where the time goes.

Aura messages can be delivered locally or remotely. In the local case (within a zone), a message is formed on the stack and delivered immediately to any receivers by calling their receive methods. In the remote case (between zones), a message is formed on the stack but copied to a FIFO buffer. The buffer is read by the receiving zone and delivered to the receiving object. (Calling a receiver's receive method directly would introduce synchronization problems as discussed earlier.)

Both the attribute/value and remote-procedure-call styles of messages rely on a hash table to translate either the attribute or the procedure name into a descriptor that can be used to access the object. Aura I uses a preprocessor to associate small integer values with attributes. This saves space (only a 32-bit identifier is transmitted rather than the string name) and makes table lookup faster (rather than computing a hash function from a string name, the hash value is simply the low-order bits of the 32-bit attribute identifier). Thus, one implementation decision we can compare is whether it is worth the trouble to map names to numbers to make lookups run faster.

Another design difference is that Aura I uses descriptors to specify the size and offset of attributes within objects. The message handler routine delivers data by copying directly into the receiving object, avoiding a method call. In Aura II, the descriptor is a function pointer; the message handler calls the function, which unpacks the message and then calls an object method. Thus, the second implementation choice we can assess is whether it is faster to interpret an object description, writing data directly into the object, or to call a method to accomplish the same result.

To measure message passing performance, a simple benchmark program sends short messages from a sender object to a receiver object. The receiver then sends a single reply, and the cycle repeats an arbitrary number of times. By sending many messages at once, the context switching time (the time it takes the operating system to switch threads and begin executing another zone) is amortized over many messages. This is important because the context switch time is greater than total time to send and receive a message!

Table 1 shows the CPU time to send one message in Aura I and Aura II, to a local object or a remote object. The message operation simply sets a 32-bit integer value in an object. These tests were conducted with compiler optimization enabled under Red Hat Linux on an IBM Thinkpad A21p laptop computer (a Pentium III running at 693MHz, slow by current standards).

Table 1. Message send times for Aura I and Aura II implementations.

Message Send Time	Aura I (attribute/value)	Aura II (remote procedure call)
Intra-zone	0.49µs	0.50µs
Inter-zone	1.02	1.15

As can be seen in the table, Aura I is about the same speed as Aura II in the intra-zone case and slightly faster in the inter-zone case. Run-time profiling was used to examine where the execution time was spent. It turns out that the technique of attribute identifiers (rather than string names) saves a substantial amount of time in the Aura I implementation. In contrast, Aura II spends approximately 100 cycles computing a simple hash function from the method signature.

On the other hand, decoding a message operation and copying data directly to or from an object is significantly slower than calling a function to accomplish the same result. Thus, Aura II gains some speed by having all object access through function pointers and method calls.

A third implementation difference is simply the size of the messages. Whereas Aura I sends a 32-bit attribute identifier, Aura II transmits a signature consisting of a signature length (8 bits), parameter types (4 bits each), and the method name (8 bits per character). Inter-zone message passing requires that messages be copied into a FIFO buffer, and in the benchmark study, this buffer is much larger than even the level-2 cache, so messages must be written to primary memory. It appears that the message length makes a significant difference in this case.

In the Aura I implementation, type-checking was disabled for this study because Aura I's run-time type checking is not implemented efficiently. A careful implementation might add 0.05 µsec to

either message passing time. Also, note that remote procedure calls can deliver multiple values in a single message, so they become more efficient as more data is passed.

This benchmark study gives us some useful information about message passing. First, general-purpose inter-thread message passing can be very efficient, with an end-to-end, type-checked remote procedure call in less than 1000 machine cycles. Secondly, one key to efficiency is to compile special-purpose code so that message handling is a matter of finding a function pointer and calling the function. In Aura, message-handling functions are generated by a preprocessor, without which any message passing scheme would be very tedious. Third, another optimization is to resolve attributes or function signatures into small integers to avoid computing hash functions on strings and to make messages shorter. The tradeoff, however, is that users must then use a preprocessor to assign numbers to names.

These same issues are relevant in other systems such as OSC [21], MAX [20, 23], and SuperCollider [15]. For example, OSC performs searching and pattern matching rather than direct method lookups, and it uses path names that, as strings, are likely to be longer than a compact numbering scheme. These choices will have a big impact on performance. I hope that this discussion will help other system designers better understand the range of possible designs and their performance implications.

6. AUDIO PROCESSING

One difficulty in Aura I is creating complex configurations of objects (unit generators) for audio processing. Audio processing support in Aura has undergone many evolutionary changes. Meanwhile, a variety of related systems have been created and are worth studying for a more complete understanding of audio processing architectures. [2-4, 13, 14, 17, 19, 22]

Unlike visual programming languages such as MAX-MSP [18], where audio "patches" tend to be static, Aura patches can be constructed and modified at run-time. While this seemed to be an important goal, it turned out to be very difficult for users. In the first implementation, forgetting to set the input of an object would cause a crash. This problem was solved by initializing all audio inputs to a "zero object," but this usually meant that every input was connected twice, first to zero, then to the correct source of samples.

To avoid dangling pointers when objects are deleted, audio connections in Aura I are doubly-linked. Deleting an object automatically disconnects any connected objects, but this adds a lot of overhead. Aura II adopts a simpler approach: all audio processing objects are reference counted, and no audio object is deleted explicitly. Therefore, there cannot be any "dangling" pointers to deleted objects. Furthermore, with remote procedure calls, we can now design initialization methods where the caller specifies everything necessary to properly create and initialize an object. This reminds the user to fully initialize objects and eliminates mysterious behavior when an input is left unconnected.

Another thing we learned using Aura is that programs that patch together many unit generators are very difficult to read. Much better would be a functional notation in which the structure of the code matches the tree structure of the audio computation graph as in Nyquist [7] or SuperCollider [15]. This was achieved using a scripting language developed for this purpose [6]. However, even this approach is not ideal. It takes quite a few messages to patch together many unit generators, and since the result is a network of separate, anonymous objects, debugging can be difficult.

After creating several compositions that included audio processing, I realized that, while a certain amount of run-time configuration of audio is useful, most of the time one wants to manipulate a whole collection of unit generators at once. I was beginning to build rather complex "instruments" that compiled as single Aura objects rather than patching together many simple objects at run time.

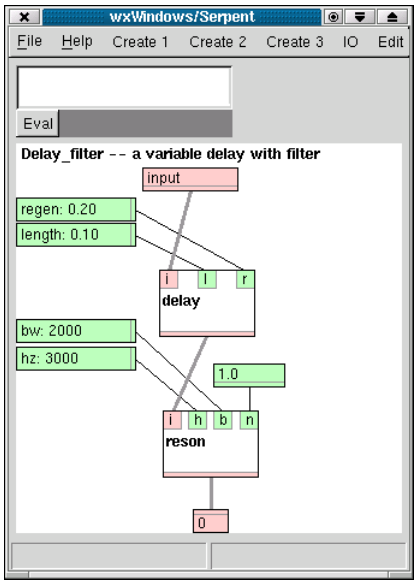

Figure 3. Aura II instrument editor. Audio inputs are at the top, outputs are at the bottom, and scalar control parameters are at left. The editor automatically selects optimal unit generator implementations based on the input and output signal types, which can be audio rate, block rate, or constant.

To support this style of programming, I developed a graphical patch editor (see Figure 3) to help create instrument objects. The patch editor chooses efficient implementations of unit generators based on whether the inputs and outputs run at the sample rate, the control rate (one sample per 32-sample audio block), or are constant values settable by messages. After a patch is completed, the editor automatically generates a C++ implementation and modifies the user's "makefile" so that the new instrument will be compiled and linked into the current application. (This mixed use of graphical programming front-end and C++ compiler back-end is also found in Open Sound World. [3]) Future enhancements will generate a graphical user interface for the instrument to facilitate testing and debugging.

7. SUMMARY AND CONCLUSIONS

It is often stated that computers are universal, that *any* sound or image is possible, and that the only limit is our imagination. Most will agree that in practice, software tools often lead creators along the path of least resistance. Just as there is pianistic music, there is csound music, MAX-MSP music, and Sonic Foundary Acid music, to name just a few. While this not necessarily a bad thing (who would say the piano is a bad idea?), we should strive for a balance between generality and power. Generality enables systems to address many problems, while power makes them easy to use.

Aura's design stresses generality. It offers an open-ended platform that provides real-time scheduling and communication support for a variety of media. Aside from a signal-processing sub-system, Aura says very little about how programs are structured. The ability to "plug in" new components and control them via an efficient messaging system has enabled a variety of applications to be written in Aura. I hope that these experiences and techniques will encourage and assist others to make software that expresses their artistic intentions effectively.

8. ACKNOWLEDGMENTS

This article summarizes lessons learned over many years, often working with students and colleagues who contributed their own experience and programming skills. In particular, Dean Rubine played a key role in defining Aura, and Eli Brandt built the first audio sub-system, which, aside from some high-level reorganization, is remarkably similar to its original form. Although this particular work is solely that of the author, systems on which this work is based received support from many sources, particularly the National Science Foundation, IBM, and Intel.

9. REFERENCES

[1] Bennett, S. *Real-Time Computer Control, An Introduction*. Prentice Hall, New York, 1994.

[2] Burk, P. JSyn - A Real-Time Synthesis API for Java. In *Proceedings of the 1998 International Computer Music Conference*, (Ann Arbor, Michigan, 1998). ICMA, San Francisco, 1998, 252-255.

[3] Chaudhary, A., Freed, A. and Wright, M. An Open Architecture for Real-Time Music Software. In *Proceedings of the 2000 International Computer Music Conference*, (Berlin, 2000). ICMA, San Francisco, 2000, 492-495.

[4] Cook, P.R. and Scavone, G. The Synthesis Toolkit (STK). In *Proceedings of the 1999 International Computer Music Conference*, (Beijing, China, 1999). International Computer Music Association, San Francisco, 1999, 164-166.

[5] Dannenberg, R.B. The CMU MIDI Toolkit. In *Proceedings of the 1986 International Computer Music Conference*, (The Hague, The Netherlands, 1986). International Computer Music Association, San Francisco, 1986, 53-56.

[6] Dannenberg, R.B. A Language for Interactive Audio Applications. In *Proceedings of the 2002 International Computer Music Conference*, (Goteborg, Sweden, 2002). ICMA, San Francisco, 2002, 509-515.

[7] Dannenberg, R.B. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal*, *21*, 3 (Fall, 1997), 50-60.

[8] Dannenberg, R.B. Real-Time Software Synthesis on Superscalar Architectures. In *Proceedings of the 1992 International Computer Music Conference*, (San Jose, CA, 1992). ICMA, San Francisco, 1992, 174-177.

[9] Dannenberg, R.B. Software Design for Interactive Multimedia Performance. *Interface - Journal of New Music Research*, *22*, 3 (August, 1993), 213-228.

[10] Dannenberg, R.B., Bernstein, B., Zeglin, G. and Neuendorffer, T. Sound Synthesis from Video, Wearable Lights, and 'The Watercourse Way'. In *Proceedings: The Ninth Biennial Symposium on Arts and Technology*, (New London, Connecticut, 2003). Connecticut College, 2003, 38-44.

[11] Dannenberg, R.B. and Neuendorffer, T. Sound Synthesis from Real-Time Video Images. In *Proceedings of the 2003 International Computer Music Conference*, (Singapore, 2003). ICMA, San Francisco, 2003, 385-388.

[12] Dannenberg, R.B. and Rubine, D. Toward Modular, Portable, Real-Time Software. In *Proceedings of the 1995 International Computer Music Conference*, (Banff, Canada, 1995). ICMA, San Francisco, 1995, 65-72.

[13] Dechelle, F., Borghesi, R., Ceccco, M.D., Maggi, E., Rovan, B. and Schnell, N. jMax: a new JAVA-based editing and control system for real-time musical applications. *Computer Music Journal*, *23*, 3 (Fall, 1998), 50-58.

[14] McCartney, J. A New, Flexible Framework for Audio and Image Synthesis. In *Proceedings of the 2000 International Computer Music Conference*, (Berlin, 2000). ICMA, San Francisco, 2000, 258-261.

[15] McCartney, J. SuperCollider: A New Real Time Synthesis Language. In *Proceedings of the 1996 International Computer Music Conference*, (Hong Kong, 1996). ICMA, San Francisco, 1996, 257-258.

[16] Myers, B., Giuse, D., Dannenberg, R., Zanden, B.V., Kosbie, D., Pervin, E., Mickish, A. and Marchal, P. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, *23*, 11 (November, 1990), 71-85.

[17] Pope, S.T. Siren: Software for Music Composition and Performance in Squeak. In *Proceedings of the 1997 International Computer Music Conference*, (Thessaloniki, Greece, 1997). ICMA, 1997.

[18] Puckette, M. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, *15*, 3 (Fall, 1991), 68-77.

[19] Puckette, M. Using Pd as a score language. In *Proceedings of the 2002 International Computer Music Conference*, (Goteborg, Sweden, 2002). International Computer Music Association, San Francisco, 2002, 184-187.

[20] Puckette, M. and Zicarelli, D. *MAX Development Package*. Opcode Systems, Inc., 1991.

[21] Wright, M. and Freed, A. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, (Thessaloniki, Greece, 1997). ICMA, San Francisco, 1997, 101-104.

[22] Wyse, L. A Sound Modeling and Synthesis System Designed for Maximum Usability. In *Proceedings of the 2003 International Computer Music Conference*, (Singapore, 2003). ICMA, San Francisco, 2003, 447-451.

[23] Zicarelli, D. An Extensible Real-Time Signal Processing Environment for Max. In *Proceedings of the 1998 International Computer Music Conference*, (Ann Arbor, MI, 1998). ICMA, 1998, 463-466.

[24] Zicarelli, D. M and Jam Factory. *Computer Music Journal*, *11*, 4 (Winter, 1987), 13-23.