Combining Visual and Textual Representations for Flexible Interactive Audio Signal Processing[*]

Roger B. Dannenberg

School of Computer Science, Carnegie Mellon University
dannenberg@cs.cmu.edu

Abstract

Interactive computer music systems pose new challenges for audio software design. In particular, there is a need for flexible run-time reconfiguration for interactive signal processing. A new version of Aura offers a graphical editor for building fixed graphs of unit generators. These graphs, called instruments, can then be instantiated, patched, and reconfigured freely at run time. This approach combines visual programming with traditional text-based programming, resulting in a structured programming model that is easy to use, is fast in execution, and offers low audio latency through fast instantiation time. The graphical editor has a novel type resolution system and automatically generates graphical interfaces for instrument testing.

1 Introduction

Flexible audio signal processing is a key ingredient in most interactive computer music. In traditional signal-processing tasks, audio processes tend to be static. For example, a digital audio effect performs a single function and the internal configuration of processing elements is fixed. Non-interactive computer music languages, in particular the Music *n* languages (Mathews 1969), introduced the idea of dynamically instantiating "instruments" to reconfigure the processing according to timed entries in a score file. In real-time interactive music systems, software must be even more dynamic, handling different media, performing computation on both symbolic and audio representations, and reconfiguring signal computation under the control of various processes.

A popular approach to building interactive software is that of MAX MSP (Puckette 1991) and its many descendents. This visual programming language allows the user to connect various modules in a directed graph. This representation is very close to the "boxes and arrows" diagrams used to describe all sorts of signal processing systems, and it can be said with confidence that this representation has withstood the test of time. However, visual representations have problems dealing with dynamic structures. It is difficult to express dynamic data structures or to reconfigure processing graphs using MAX-like languages.

The main alternative is to use text-based languages. *SuperCollider* (McCartney 1996) is a good example of a text-based system for interactive computer music. Text-based languages at least offer the possibility of reconfiguring audio computation graphs, which is an attractive possibility for music performance. The ability to decide at any time to apply an effect or to change the way parameters are computed can make programs more versatile while minimizing the computation load that would occur if all options were running all the time.

Aura is another text-based (until recently) programming environment for interactive music performance systems. Aura began almost 10 years ago as an object-oriented generalization of the CMU MIDI Toolkit. (Dannenberg 1986) The Aura object model provides a natural way to create, connect, and control audio signal processing modules. Aura was designed with audio processing in mind, and it therefore offers multiple threads and support for low-latency real-time audio computation.

The goal of this paper is to discuss software architecture issues associated with interactive audio, especially in the light of years of experience creating interactive compositions with Aura. In particular, the latest version of Aura, which I will call *Aura 2,* combines a visual editor with a text-based scripting language in an effort to obtain the best of both worlds of visual and text-based programming languages.

Aura 2 uses a visual editor to combine unit generators into (potentially) larger units called *instruments*. At run-time, under program control, instruments can be dynamically allocated and patched to form an audio computation graph. The periodic evaluation of this graph is scheduled along with other events that can update instrument parameters, create new instruments, and modify the graph. A remote-procedure-call facility allows I/O, graphics, and control computation to run outside of the time-critical audio-computation thread.

Section 2 describes some related work, and Section 3 discusses design alternatives and the choices made for Aura 2. Section 4 describes the Aura Instrument Editor, a new Aura component that allows users to edit signal-processing algorithms graphically. This is followed by a description of how instrument designs are incorporated into Aura programs and debugged there. Section 6 discusses the range of programming styles supported, from static to very dynamic, and the current status and future work are described in

*Published as: Roger B. Dannenberg, "Combining Visual and Textual Representations for Flexible Interactive Signal Processing," in *The ICMC 2004 Proceedings*, San Francisco: The International Computer Music Association, 2004.

Section 7. Finally, Section 8 presents a summary and conclusions.

2 Related Work

There are many real-time audio processing frameworks and systems, but space limitations prevent a detailed review. Once the basic problems of audio I/O are surmounted (using systems like PortAudio (Bencina and Burk 2001) or RtAudio (Scavone 2002)), the next big issue is how to modularize DSP algorithms for reuse. The standard approach is the unit generator concept, introduced in Music *n*, in which signal processing objects have inputs, outputs, some internal state, and a method that processes one unit of input to produce outputs.

Systems vary according to how unit generators are assembled and managed at run time. The simplest approach is through mainly static graphs as in MAX-like languages. (Chaudhary, Freed, and Wright 2000; Dechelle et al. 1998; Puckette 1991; Puckette 1997; Puckette 2002a; Zicarelli 1998) Kyma (Scaletti and Johnson 1988) might also belong in this category because it uses precompiled graphs.

A more elaborate approach is based on Music *n* in which a group of unit generators (called a *patch* or an *instrument*) is instantiated dynamically. A feature of this approach is that instruments add their outputs to a global buffer. The instrument can be deallocated without leaving behind any "dangling" pointers to deleted objects. The NeXT Music Kit (Jaffe and Boynton 1989) and csound (Vercoe and Ellis 1990) adopted this approach, and to a large extent, this is also the basis for managing instruments in SuperCollider (McCartney 1996). Similar principles are also at work in ChucK (Wang and Cook 2003) insofar as audio "shreds" can be started or stopped independently.

Another way to structure computation is to use the tree-like structure of nested expressions, e.g. *mult*(*osc*(), *env*()), to denote a corresponding graph structure of unit generators. This sort of implicit connection is found in SuperCollider, STK (Cook and Scavone 1999), Sizzle (Pope and Ramakrishnan 2003) and Nyquist (Dannenberg 1997).

Other systems, including CLM (Schottstaedt 1994), JSyn (Burk 1998), and Aura 1 (Dannenberg and Brandt 1996) impose even less organization upon programmers, allowing arbitrary patching and repatching at run-time.

3 Design Issues

The premise of this work is that current systems suffer because they focus too narrowly on a single paradigm. Graphical editing systems are simple to learn and facilitate rapid experimentation, but patches are hard to reconfigure under program control. Text-based languages offer more flexibility, but lose some of the intuitive feel and debugging support of a visual representation.

Beyond the question of "graphical vs. textual," there is a deeper issue that I call "static vs. dynamic": to what extent are interconnections known at design time vs. run time? When connections are known in advance, they can offer more efficient compilation, more context for debugging, and faster instantiation at run time. On the other hand, when connections are made at run time, configurations and reconfigurations can be generated interactively under computer control. This is certainly more flexible.

Aura 2 includes a new audio subsystem that supports a range of processing models, from fully static audio processing graphs to fully dynamic ones. Static graphs of unit generators are configured using a graphical editor (see Figure 1) to form *instruments*, members of the Aura *Instr* class. These instruments can be flexibly interconnected at run time under program control.

Experience with fully dynamic and reconfigurable graphs in the first version of Aura indicates, not surprisingly, that most audio processing designs are mainly static. While there may be good reason to allow and even promote dynamic changes to audio signal processing graphs, in practice, most connections are set up when an instrument is created and stay that way for as long as the instrument exists. Often, boxes-and-arrows diagrams are used at the design stage, even when this must be translated to a textual implementation, so Aura 2 supports this style of working by providing a graphical patch editor.

By organizing collections of unit generators into instruments, we benefit in several ways. First, we do not pay for the overhead of dynamic patching when statically allocated unit generators executed in a fixed order will suffice. Second, when problems occur, symbolic debuggers tend to provide more useful information when unit generators have names and exist in the context of a larger object than when they are dynamically allocated individually on the heap. Third, in-lining unit generators and other compilation tricks can improve the performance of the instrument model. Finally, and most importantly, users can reason about their own code more easily when more of the design is static.

The latest version of SuperCollider reflects some of these thoughts and design principles; in particular, SuperCollider 3 instruments are compiled into a single object as opposed to composing unit generators at run time. The justification is in part to avoid heavy computation in order to instantiate a new instrument. (McCartney 2002)

Aura 2 also pays attention to the problems that arise when graphs are more dynamic. As graphs are reconfigured, Aura automatically calculates the instrument execution order so that an instrument's outputs are always up-to-date before another instrument reads them. Instruments have infinite fan-out – the output of one instrument can be shared by any number of other instruments. Because instruments can be referenced by many others, reference counting is used to delete an instrument only after there are no more references to it. This makes it unnecessary to explicitly delete modulation sources and other signal sources that may have been attached to a sound generator that has terminated. Reference counting also eliminates the problem of deleting an instrument too early. Finally, Aura 2 uses global names for instruments (and all Aura objects) and a remote-procedure call system. Instruments can be fully controlled from processes running asynchronously with respect to the time-critical audio processing loop.

4 The Aura Instrument Editor

The Aura Instrument Editor (AIE) is a graphical editor for combining unit generators into instruments. (See Figure 1.) This is conceptually very similar to a number of programs used for various systems (Bate 1990; Helmuth 1990; Minnick 1990), so I will try to focus only on some of the interesting features.

One feature is an attempt to integrate the visual editor with text-based programming. When the user creates a new instrument, the editor can automatically update the user's Makefile and write scripting language functions to create an instance of the instrument. In this way, new instruments are automatically integrated into the user's program. In addition, the editor can automatically process unit generator source code to add new unit generators to the AIE menus.

Although not shown in Figure 1, when the mouse is positioned over a unit generator, a description appears on the screen. Similarly, the full parameter name appears when the mouse is moved over an input.

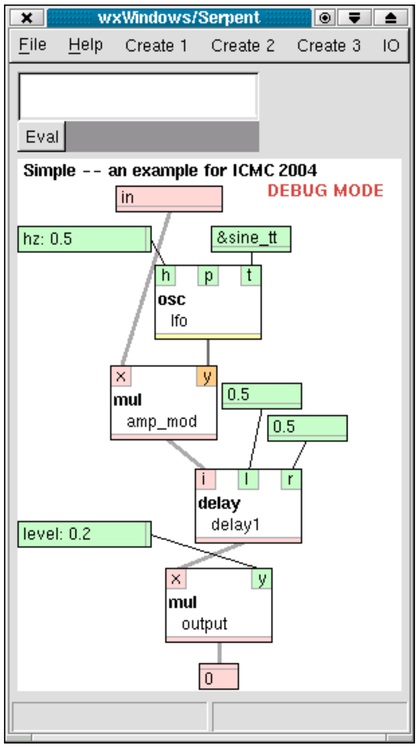

Figure 1. The Aura Instrument Editor. Instrument inputs and outputs are color coded to indicate signal types.

4.1 Type Resolution

For efficiency, Aura supports various types of signals, and the editor helps the user by automatically selecting suitable types and compatible unit generators. The types are:

- Audio-rate ("a") signals are streams of floating-point numbers processed one block at a time (typically 32 samples per block).
- Block rate ("b") signals are computed synchronously with audio, but consist of only one sample per block (similar to csound's *k-rate* signals).
- Constant rate ("c") unit generator inputs remain at a constant value until changed by a message.
- Interpolated ("i") inputs accept a block rate signal and interpolate it to audio rates internally.

In the AIE, audio rate, block rate, and constant rate signal inputs and outputs are color coded using red, yellow, and green, and lines representing signals are coded using different widths. The user selects unit generators by their generic name, e.g. "osci" is an interpolating oscillator with frequency, initial phase, and wave table parameters. It is the editor's job to select the correct implementation, which may be one of *Uosci_acc_a*, *Uosci_bcc_a*, *Uosci_ccc_a*, *Uosci_bcc_b*, *Uosci_ccc_b*. Note how the names encode the input types and the output type.

To determine the unit generator types, the editor must label the signal paths with their types. Among the constraints are:

- Signal (connector) types must match the type of the input to which they are connected;
- Signal types must match the type of the output to which they are connected;
- Only one block rate or audio rate signal can be connected to a given input;
- Unit generator inputs that do not have any incoming signal must be of type constant-rate.

These constraints do not always lead to a unique solution, so a heuristic is used to guess what the user wants: When in doubt, prefer block-rate over audio-rate. In addition, the user can "pin" the type of any signal to the preferred rate to override the editor's choice.

This type resolution problem is NP-complete (using two types to represent Boolean values, it is straightforward to reduce type resolution to circuit satisfiability), so a general solution would require backtracking and could take time exponential in the size of the graph. In practice, labeling "real" graphs is not so difficult. I created an iterative algorithm that runs in O(n^2) time, where n is the total number of graph nodes and edges. The algorithm can fail to find a valid assignment of types, but in this unlikely event, the user can always label a signal explicitly. The gist of the algorithm is in Figure 2. Steps are executed in order unless otherwise specified. The algorithm terminates when all steps are executed with no changes to the labeling.

This algorithm will iterate as long as progress is being made. Since there is no backtracking or "unlabeling," the algorithm will eventually halt. If all inputs, outputs, and signals are labeled, the algorithm succeeds. When a valid labeling is not possible, the offending unit generators and/or signals are labeled in bright red. Usually this means that one or more inputs or outputs are unconnected.

Typically, this algorithm terminates in just a few iterations, which is fast enough that the entire graph can be relabeled after every editing operation. Thus, the user receives instant visual feedback about the types of all signals.

Initialization. For each unit generator in the graph, there is a set of possible implementations; each implementation specifies the type of each input and output. Begin by labeling the type of each unit generator input and output as –unknown“ and label the type of each signal as –unknown“ Signals that are explicitly labeled by the user are marked accordingly, and inputs with no attached signals are labeled as –constant rate.“

Propagate types. Examine the inputs and outputs at the end-points of each signal. If the type at one end is known, set the signal type and the input or output at the other end-point to the same type. Note that progress has been made.

Check for ambiguity. For each unit generator, enumerate the set of implementations that match the known input and output types. For each unknown input or output type, do all of the implementations specify the same type? If so, then set the input or output type accordingly. For example, if there are three possible implementations, but all three have an audio-rate output, the type of the output is –audio rate.“ If a type is resolved, note that progress has been made.

Iterate. As long as progress is being made, repeat –**Propagate types**“ and –**Check for ambiguity**.“

Fan-in. If an input has more than one connected signal, set its type to –constant rate“ and **Iterate**.

Prefer block rate heuristic. If the possible implementations for a unit generator include both –audio rate“ and –block rate“ for the output, label the output type as –block rate“ and **Iterate**.

Figure 2. The type resolution algorithm.

4.2 Buffer Allocation and Performance

After the user creates a valid instrument, the editor can generate an implementation in C++. The implementation declares each unit generator and defines a method to perform one computation step that invokes each unit generator. A topological sort is performed on the graph to order the unit generators so that signals flow from input to output in a single pass. Buffers are allocated to hold intermediate results (the signals connecting unit generators), and buffers are reused where possible to minimize storage. Since buffers are temporary, they are allocated on the stack.

A common assumption is that buffer organization is important for good cache performance in signal processing applications. By modifying the AIE, we can evaluate the impact of different buffer allocation policies. I compared the performance of a simple patch using optimal buffering and with no buffer optimization (each signal has its own buffer of 32 floats). The patch is a simple FM instrument with vibrato and a resonant filter. It has 6 audio-rate unit generators and 10 block-rate unit generators, four of which are envelope generators.

I modified Aura's audio output object to time how long it takes to compute a given number of sample blocks (without writing them to the DAC). Interestingly, there is no measurable difference between the optimized and unoptimized buffer allocation policy. In tests with from 1 to 128 copies of the instrument using either optimized or unoptimized buffer allocation, the run time per computed block only varies by 0.5 percent.

I also measured the performance using Aura's old unit generator model where each unit generator is dynamically allocated and patched to form a graph at run time. In this case, the performance is about 16% slower than the new model. This is because of the overhead to access a unit generator includes testing to see if each input is valid, and if not, calling a unit generator to provide the input, then making a C++ virtual method call to run the unit generator.

The old unit generator model allocates a buffer on the heap for each unit generator output. Figure 2 shows a graph of performance as more copies of the graph are executed. The graph is essentially flat until around 64 copies, after which the execution time increases. 128 copies corresponds to about 100KB of buffer data (not to mention other object data), so it seems plausible that at this point, buffer writes are forcing object data out of the 256KB secondary cache and causing cache misses. Even then, the penalty is less than a factor of two, in part due to the prefetching behavior of the cache. (Dannenberg and Thompson 1997) Furthermore, this particular processor is only able to run about 64 voices in real time, so any performance degradation with more voices would not show up in practice.

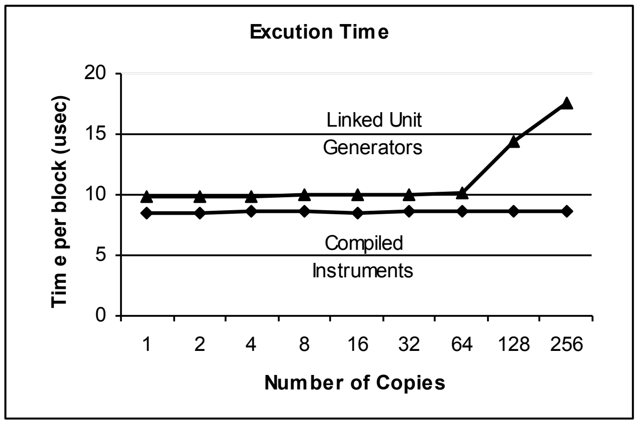

Figure 3. The instrument model with optimized buffer allocation and sequentially executed, in-lined unit generators performs slightly better than a graph of unit generators linked and accessed via pointers.

5 Debugging Support

When a new instrument is created, it is often necessary to build some scaffolding to test and debug the instrument. Because the Aura Instrument Editor knows all the inputs and outputs of an instrument design, it can automatically create a graphical user interface to streamline the testing phase. Figure 3 shows the control panel created for the instrument named "Simple" in Figure 1. The AIE also adds the instrument to a "Test" menu, so when the user runs Aura, control panels are immediately available for all instruments.

To test *Simple*, the user selects "Simple" from the "Test" menu and the window shown in Figure 4 appears. At the upper left corner is a "play" button. This creates an instance of the *Simple* instrument and connects it to the audio output. Below the play button, you can see labels that correspond to the instrument inputs: "in," "level," and "hz".

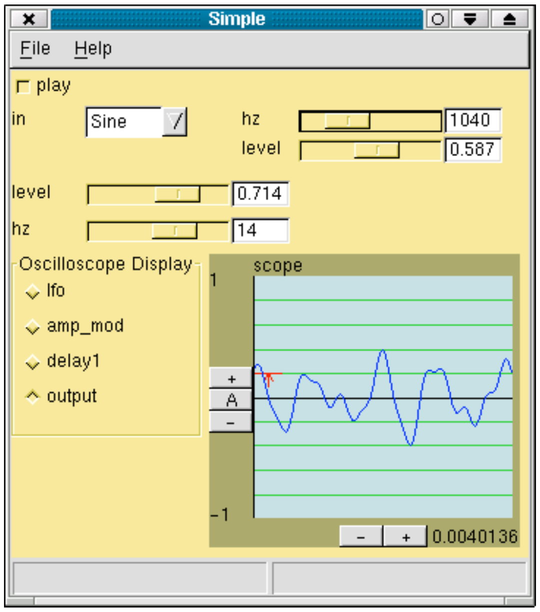

Figure 4. Automatically generated control panel to test the "Simple" instrument shown in Figure 1.

Audio inputs such as "in" have fairly elaborate controls. The audio source can be silence, a sinusoid (as shown), white noise, an audio file, or real-time audio input. A slider is provided to set the sinusoid's frequency, and a level control sets the input level.

Constant and block-rate inputs are controlled by sliders as shown next to "level" and "hz." The range of the slider defaults to 0 to 1, but the user can change the range using the AIE.

At the bottom of the control panel, there is an oscilloscope display that can be used to probe the signals within the instrument. Notice the set of buttons labeled with unit generator names ("lfo," "amp_mod," etc.) These select a signal for display and correspond to the unit generator names in Figure 1. Figure 4 shows the instrument output on the oscilloscope. In steady-state, this simple instrument produces very boring displays, so I wiggled the frequency of the sinusoidal input just before the screen capture.

At the left edge of the oscilloscope signal display, there is a short horizontal line segment above an upward-pointing arrow. This indicates that the oscilloscope begins a sweep when the signal crosses the indicated threshold in the upward direction. The threshold and direction are set by a mouse gesture: mouse down on the desired threshold, and drag in the direction of the threshold crossing. This is a simple refinement, but it is quite handy in practice.

The oscilloscope display is enabled by an option in the AIE (notice the label "DEBUG MODE" at the upper right of Figure 1). When the debug mode is selected, all signal buffers become instrument outputs. An instrument of class *Probe* is used to collect samples in the high-priority Aura audio thread and send them via Aura's remote procedure call (Dannenberg 2004) to the *Oscilloscope* object in the user-interface thread. The *Probe* object collects samples only on request to avoid overloading the interface and overflowing buffers.

6 Dynamic Configuration

So far, I have emphasized a more static instrument model, and nothing has been said about dynamic alterations to the signal computation graph. Aura 2 instruments can be connected and disconnected from one another at run time. For each signal input, a "set" method is generated. The user can make a remote procedure call from anywhere to connect the output of one instrument to the input of another. (Keep in mind that an "instrument" is Aura's generic term for any source of audio or control signals. Instruments can serve as controllers, digital audio effects, synthesizer voices, sound file players, mixers, filters, etc.) Programming can be done in C++ or Serpent, Aura's real-time scripting language. (Dannenberg 2002)

For example, to create and play an instance of *Simple* with input from *My_source*, one can write:

simple = *simple_create*(*my_source_create*())
aura_play(*simple*)

To dispose of the input and replace it with audio input, one writes:

simple_set_in_to(*simple*, *audio_io_aura*)

To delete this instance of *Simple*, one disconnects it from the audio output object:

aura_mute(*simple*)

To create several instances of *Simple* at random times, first define a function to create, play, and end a *Simple*:

```
 def simple_player():
     var simple = simple_create(audio_in_aura)
     aura_play(simple)
     // optional parameters say to end after 3 seconds:
     aura_mute(simple, 0, after(3))
```

Then, we can use *cause* to invoke possibly overlapping copies of *simple_player* after random delays:

```
for i = 0 to 5:
    cause(random() * 10, 'simple_player')
```

One could also save references to these instances, allowing interactive, ongoing update messages to control them. This sort of dynamic configuration and control is very difficult to achieve in purely visual music programming languages.

To avoid dangling pointers to instruments, reference counting is used. By default, instruments are deleted when no longer attached to the graph. Thus, if you attach a special modulator to a sound and the sound is deleted, the attached modulator will also be deleted. Alternatively, the reference count can be incremented by the creator of the instrument. The instrument will then be retained for reuse as long as the creator holds the reference.

6.1 Unit Generator Order of Evaluation

Aura uses the instrument graph structure to determine the order of execution of the instruments. If the graph changes, the user does not have to worry about execution order because it is implicitly recomputed on every graph traversal. In contrast, SuperCollider places order largely under manual control, MAX compiles a fixed order for the entire (static) graph, and CSL/Sizzle requires special fan-out objects to allow a signal to be shared by multiple readers.

Aura's algorithm is described here using pseudo-code. Each instrument object computes samples by processing samples from other instruments, and computed samples are retained in the instrument's output buffer(s). References to other instruments are stored in the *inputs* array. Instruments are visited using a form of topological sort: a sample block counter keeps track of when it is necessary to calculate samples, which are calculated just before the first reader needs them. The graph traversal is started by calling *run* in the audio output object, and *run* operates recursively to traverse the graph. The pseudo-code for *run* is shown in Figure 5.

```
method run(newcount):
    foreach input in inputs:
        if input.count < newcount: input.run(newcount)
    real_run() –compute the samples for this ugen,
                reading samples from inputs
    count = newcount –this instrument is up-to-date
```

Figure 5. The run method is used to traverse the instrument graph in the data-dependent order.

This algorithm adds a simple compare and a store to the overhead of traversing the graph, which is negligible given that this cost is amortized over all the unit generators within an instrument. Figure 6 illustrates the traversal path of a small graph. Within an instrument, the configuration is fixed, so the AIE-generated *real_run* method consists of a sequential execution of the instrument's unit generators.

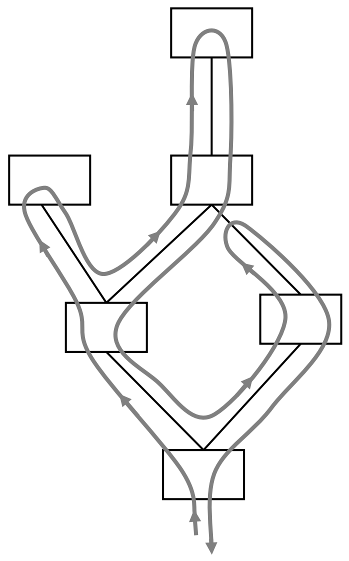

Figure 6. Traversal order of the instrument graph, shown in gray curve, is computed on the fly.

6.2 Support for Multiple Styles

My goal is to support various approaches to audio processing. The very static style of MAX-like languages is easily accomplished by assembling a graph of instruments and leaving it in place. The more dynamic Music *n* and SuperCollider style, where notes or sounds are dynamically instantiated, play to completion, and are deleted, are simple to implement using timed messages to create and delete sounds and the reference counting scheme to clean up. More general schemes are supported by giving the user the full ability to repatch the graph at run time.

7 Current Status, Future Work

Aura 2 is running under Linux, using PortAudio (Bencina and Burk 2001) and PortMidi (http://www.cs.cmu.edu/~music/portmusic) for I/O, Serpent (Dannenberg 2002) as a real-time scripting language, and wxWidgets (formerly wxWindows, http://www.wxwindows.org) for the graphical user interface. The Aura Instrument Editor is implemented in Serpent. Source is freely available, but Aura is not yet packaged as easy-to-use, well-documented software.

Many enhancements are possible. The Aura Instrument Editor would benefit from more direct manipulation as opposed to menus and pop-up forms. It also needs an undo facility and scrolling, and it would be nice to represent at least multiplication and addition with special small icons.

The editor does not currently support any kind of hierarchical description. It would be nice to allow structured elements such as Aura instruments to be inserted in place of a unit generator. This would be especially useful in multichannel instruments where two or more copies of the same graph are needed.

It might be desirable to extend Aura signal types to include spectral frames and other types. While most systems avoid this by coercing spectra into signals, the type resolution system might make it more practical and even desirable to let types proliferate.

Since Aura instruments are compiled, it should not be difficult to allow users to insert expressions and function calls into their instrument designs. (Chaudhary, Freed, and Wright 2000) This could be especially useful for processing "constant rate" inputs. A constant rate input is updated with a message, and sometimes it is desirable to perform actions upon the receipt of a message.

Another optimization for signal processing is to merge the inner loops of unit generators to form larger loops. (Dannenberg and Thompson 1997) Signal values then can be passed through registers rather than memory buffers, and there is less loop overhead. The AIE could be extended to perform this sort of optimization.

The integration of scores with real-time interactive systems (Puckette 2002b) is not well-understood, and this is certainly an area for future exploration in Aura 2. Aura's new support for dynamically instantiated instruments should allow a conceptually simple mapping from score to run-time processing.

8 Summary and Conclusions

Aura 2 is a new version of the Aura platform for constructing real-time, interactive music systems. Changes have been made based on experience and lessons learned from the creation of many interactive pieces using Aura.

One lesson learned is that even when audio computation can be completely reconfigured on-the-fly, it is easier to think of the system in terms of fairly significant modules or subgraphs that are never reconfigured internally. In fact, these modules are often designed and documented using a boxes-and-arrows diagram even when the implementation uses a text-based language. The second lesson is that supporting reconfigurability requires extra layers of abstraction that ultimately make systems more complicated and difficult to debug.

With these lessons in mind, I created a graphical instrument editor to better support the programming style that seemed most natural. Two nice side effects of this approach are (1) signal processing is slightly faster due to a reduction in processing overhead, and (2) the instrument editor is able to automatically generate control panels that greatly facilitate experimentation and debugging. The editor also updates the user's Makefile, simplifying the integration of new instruments into programming projects.

A key feature of the editor is its type resolution system, which automatically deduces appropriate signal types and selects optimized unit generator implementations. By performing this resolution at design time, the user can see visually how the system is implemented, and instruments with many unit generators can be instantiated very quickly at run time, reducing system latency.

The availability of an instrument code generator facilitated a study of the effect of buffer allocation on performance. Current "wisdom" or folk lore dictates that buffer allocation is important to performance. In our experiments, buffer optimization had no measurable effect on run time when buffers are allocated temporarily on the run-time stack. When buffers are allocated statically (one per unit generator), there is performance degradation, but not in real-time operating ranges below 100% CPU utilization. Systems with larger blocks sizes (Aura uses 32-sample blocks) or faster processors without correspondingly larger caches *might* want to consider schemes for reusing buffers. If instruments combine unit generators and buffers are local (on the stack), it is unlikely that any further optimization will help.

While the graphical instrument editor facilitates the programming of fixed audio processing patches, careful attention also has been given to text-based programming support. Aura instruments automatically compute the correct execution order, reference counting facilitates the sharing and automatic deletion of instruments, and Aura's remote procedure call mechanism makes it easy to configure audio processing from a process outside of the time-critical audio processing loop. By compiling unit generator graphs, dynamic instantiation and deletion of audio processing objects is faster.

In conclusion, real-time audio processing has become an essential part of interactive computer music. It is important that we understand the degree to which our languages and systems determine what we create. Coming from the other direction, it is important to design our languages and systems to support rather than limit the creative process. Towards this goal, Aura 2 offers an easy to use editor for signal processing, combined with a flexible run-time system that supports dynamic, on-the-fly reconfiguration.

9 Acknowledgments

Eli Brandt implemented Aura's first audio processing system, which formed the basis for this new work. I have benefited from conversations with many audio system developers, including Ron Kuivila, James McCartney, Stephen Pope, and Amar Chaudhary. Thanks to Ning Hu and Eli Brandt for helpful comments on this paper.

References

Bate, J. 1990. "UNISON - a Real-Time Interactive System for Digital Sound Synthesis." In *1990 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 172-174.

Bencina, R., and P. Burk. 2001. "PortAudio - An Open Source Cross Platform Audio API." In *Proceedings of the 2001 International Computer Music Conference*. San

Francisco: International Computer Music Association, pp. 263-266.

Burk, P. 1998. "JSyn - A Real-Time Synthesis API for Java." In *Proceedings of the 1998 International Computer Music Conference*. San Francisco: International Computer Music Conference, pp. 252-255.

Chaudhary, A., A. Freed, and M. Wright. 2000. "An Open Architecture for Real-Time Music Software." In *Proceedings of the 2000 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 492-495.

Cook, P. R., and G. Scavone. 1999. "The Synthesis Toolkit (STK)." In *Proceedings of the 1999 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 164-166.

Dannenberg, R. B. 1986. "The CMU MIDI Toolkit." In *Proceedings of the 1986 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 53-56.

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

Dannenberg, R. B. 2002. "A Language for Interactive Audio Applications." In *Proceedings of the 2002 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 509-515.

Dannenberg, R. B. 2004. "Aura II: Making Real-Time Systems Safe for Music." In *(in review)*.

Dannenberg, R. B., and E. Brandt. 1996. "A Flexible Real-Time Software Synthesis System." In *Proceedings of the 1996 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 270-273.

Dannenberg, R. B., and N. Thompson. 1997. "Real-Time Software Synthesis on Superscaler Architectures." *Computer Music Journal, 21*(3), 83-94.

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

Helmuth, M. 1990. "PatchMix: A C++ X Graphical Interface to Cmix." In *1990 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 273-275.

Jaffe, D. A., and L. R. Boynton. 1989. "An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal, 13*(2), 48-55.

Mathews, M. 1969. *The Technology of Computer Music*: MIT Press.

McCartney, J. 1996. "SuperCollider: A New Real Time Synthesis Language." In *Proceedings of the 1996 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 257-258.

McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal, 26*(4), 61-68.

Minnick, M. 1990. "A Graphical Editor for Building Unit Generator Patches." In *1990 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 253-255.

Pope, S. T., and C. Ramakrishnan. 2003. "The CREATE Signal Library ("Sizzle"): Design, Issues, and Applications." In *Proceedings of the 2003 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 415-422.

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

Puckette, M. 1997. "Pure Data." In *1997 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 224-227.

Puckette, M. 2002a. "Max at Seventeen." *Computer Music Journal, 26*(4), 31-43.

Puckette, M. 2002b. "Using Pd as a score language." In *Proceedings of the 2002 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 184-187.

Scaletti, C., and R. E. Johnson. 1988. "An Interactive Graphic Environment for Object-oriented Music Composition and Sound Synthesis." In *Proceedings of the Conference on Object-Oriented Programming Languages and Systems*. New York: ACM Press, pp. 18-26.

Scavone, G. P. 2002. "RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output." In *Proceedings of the 2002 International Computer Music Conference*. International Computer Music Association, pp. 196-199.

Schottstaedt, B. 1994. "Machine Tongues XVII: CLM: Music V Meets Common Lisp." *Computer Music Journal, 18*(2), 30-37.

Vercoe, B., and D. Ellis. 1990. "Real-Time CSOUND: Software Synthesis with Sensing and Control." In *1990 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 209-211.

Wang, G., and P. R. Cook. 2003. "ChucK: A Concurrent, On-the-fly, Audio Programming Language." In *Proceedings of the 2003 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 219-226.

Zicarelli, D. 1998. "An Extensible Real-Time Signal Processing Environment for Max." In *Proceedings of the 1998 International Computer Music Conference*. International Computer Music Association, pp. 463-466.