AURAFX: A SIMPLE AND FLEXIBLE APPROACH TO INTERACTIVE AUDIO EFFECT-BASED COMPOSITION AND PERFORMANCE

Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science

Robert Kotcher
Carnegie Mellon University
School of Music

ABSTRACT

An interactive sound processor is an important tool for just about any modern composer. Performers and composers use interactive computer systems to process sound from live instruments. In many cases, audio processing could be handled using off-the-shelf signal processors. However, most composers favor a system that is more open-ended and extensible. Programmable systems are open-ended, but they leave many details to the composer, including graphical control interfaces, mixing and cross-fade automation, saving and restoring parameter settings, and sequencing through configurations of effects. Our work attempts to establish an *architecture* that provides these facilities without programming. It factors the problem into a *framework,* providing common elements for all compositions, and *custom modules,* extending the framework with unique effects and signal processing capabilities. Although we believe the architecture could be supported by many audio programming systems, we have created a particular instantiation (AuraFX) of the architecture using the Aura system.

1. INTRODUCTION

Interactive music compositions span a wide range of organizations, intentions, and implementations. Most music in this category can be labelled "experimental," and as such calls for very open-ended and general software for development. One of the reasons for generality is to avoid falling into the trap of the "paint-by-number" approach, where only a fixed set of effects, timbres, or patches is available to choose from. In this case, the sounds become recognizable and unoriginal – what composer would want to write something that has been heard before?

As computer music has matured and more systems have become available, the "paint-by-number" approach has become more acceptable for several reasons: (1) more effects and sound processing systems are available, so the composer is not limited to a small set of choices; (2) modern computing power has enabled more flexibility, parameter choices, and other ways to customize off-the-shelf effects; (3) computer music systems are widely available and attractive to non-expert programmers. When computer music *required* considerable engineering skill, it was normal for musicians to do a considerable amount of programming, but today, many musicians find it acceptable to use relatively pre-packaged engineering solutions in order to instead focus their creativity on music composition and performance.

Our implementation of the AuraFX system follows a tradition of trying to understand the general nature of some broad class of computer music systems and make it easy to create new systems in that class. Ideally, it should allow the composer or sound engineer to accomplish tasks with a minimum of effort and a maximum of flexibility. The *architecture* that underlies AuraFX is simple: a sequence of sets of effects with adjustable levels, fade-in and fade-out times, and channel assignments. The effects have a well-defined interface with the system so that new effects can be constructed and "plugged in" to the architecture. Transitions from one set of effects to another can be sequential (e.g. advanced by a simple pedal interface or timer) or controlled by an external program.

An important issue in our design is the user's technical competence. Many composers say they use a visual programming system such as MAX MSP [1] or Pd [2] because they "don't know how to program," but even these visual programming systems require programming. Developing patches, controlling them, sequencing them, and mixing their outputs all requires programming, yet these languages have at best a limited way to organize and accomplish these programming tasks. AuraFX imposes much more structure and in return requires much less programming than even visual programming systems. In fact, the only programming available in AuraFX is the writing of effects using either a scripting language or a dataflow-like visual programming language. AuraFX has built-in mechanisms for allocating, mixing, and sequencing effects. On the other hand, AuraFX is more open-ended than an automated mixer because it is not restricted to sequential or time-based changes and it is particularly adept at dynamically scheduling and managing multiple effects which might overwhelm the processor(s) if all of the effects were set up at once on different effect-send busses.

Published as: Dannenberg and Kotcher, "AuraFX: A Simple and Flexible Approach to Interactive Audio Effect-Based Composition and Performance," in *Proceedings of the 2010 International Computer Music Conference*, San Francisco: The International Computer Music Association, (August 2010), pp. 147-152.

AuraFX has several motivations. It was originally created for a performance by the Pittsburgh New Music Ensemble (PNME) at the Edinburgh Festival Fringe. The PNME planned to use extensive audio processing but needed a system that could be rapidly reconfigured after arriving at the performance space. Although the ensemble abandoned their plans for electronics as too ambitious, the discussion forced us to think about how to create a flexible audio processing system for use by non-programmers. The second motivation is the first author's own work with Aura, an open-ended system for interactive multimedia. One of the things we noticed in using Aura is that much of the most difficult programming effort in actual compositions had to do with managing transitions: making sure effects are running and connected when needed, making soft switches rather than abrupt connections, and building interfaces not only for normal operation, but for rehearsals and debugging. Once AuraFX was started, we realized it could also be used in improvisational settings where effects can be called up at will (for example using foot pedals or algorithmic selection), and the system can be extended over time in a modular way.

2. RELATED WORK

Computer music systems and languages attempt to simplify the task of writing interactive music systems. The Music N languages [[3], [4]] introduced the concept of unit generators as powerful primitives for musical signal processing. Two important approaches for more interactive systems include MAX-like visual programming languages [[1], [2]] and text-based programming languages [[5][10]]. Jamoma [11] is an attempt to create higher-level abstractions in MAX MSP that provide audio signal processing modules with built-in user interfaces, all designed to have a common look and feel and sophisticated control such as the ability to smoothly interpolate input parameter values and respond to Open Sound Control messages. Audio Mulch [12] is another example of a programmable (through graphical patching) system based on sophisticated audio processing modules with user interfaces.

AuraFX differs from this work because it aims to avoid programming (even most graphical patching), relying instead upon built-in methods that manage parameter changes, transitions between effects, and activation/deactivation of signal processing modules. AuraFX is extensible primarily through the addition of effects which can be created using graphical patching of unit generators.

3. THE AURAFX ARCHITECTURE

AuraFX is designed around a particular computational and signal processing model consisting of *effects*, *states*, and *sequences.*

3.1. Effects

First and foremost, processing is organized around *effects*, which are algorithms for processing live audio. Figure 1 shows a signal flow diagram for an effect. Audio inputs (4 are shown) are mixed and fed into the *DSP Algorithm*. Essentially any process can be inserted here. The output of the DSP Algorithm is faded in and out using an envelope, and the signal is then panned among output channels (2 are shown). All effect outputs are summed to form the overall program outputs (not shown).

At present, effects are mono in and mono out for simplicity. Extending this to multi-channel effects would require a gain matrix to steer inputs to the effect and another matrix to route effect outputs to the outputs. This presents more of a user interface design problem than anything else.

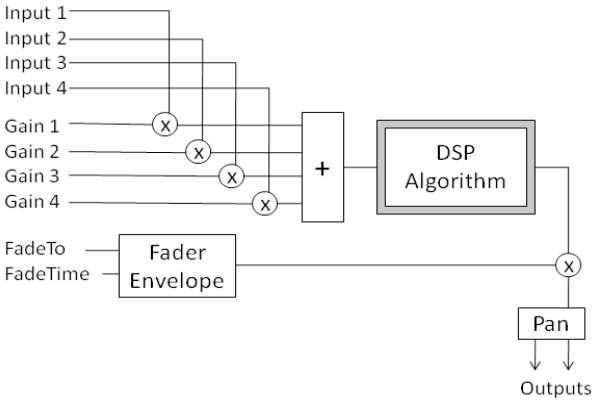

Figure 1. Signal diagram for an effect.

3.2. States

Effects are organized according to *states*. A state is a collection of effects that operate concurrently and independently. In operation, there is normally one active state that determines what effects are in use. However, when a new state is selected, there is a transition implemented by fading out the effects in the previous state and fading in the effects in the new state.

3.3. Sequences

A sequence is an ordered list of state references. A state may be referenced from multiple positions of the same sequence. References to states rather than copies of states are used so that when a state is edited in one sequence position, the changes take effect in each position where the state is used. To avoid this behavior, it is only necessary to copy the state and insert these copies into the sequence.

In a composition organized as a linear score with changing audio effects, the sequence organizes the progression from one set of effects to the next. It is also possible to ignore the linear structure and jump to any location in the sequence.

In addition to a state reference, each element of a sequence contains a set of gain and pan controls for each audio input, a global reverb level, and fade-in and fade-out times. Thus, the sequence can be viewed as "scenes" in an automated mixer.

3.4. Transitions

There are many cases where it is desirable to change the value of an effect parameter rather than replace one effect by another. For example, there can be quite a difference between changing a delay amount and cross-fading from one delay effect to another. Our system tries to reuse active effects by changing parameter values. This allows for greater sonic continuity in transitions and also increases efficiency. It takes less processing to change an effect parameter than it does to run a second effect while cross-fading.

To understand how effects are reused, let *E* be the set of active effects, that is, effects with non-zero output levels, and let *B* be the effects contained in state B. In a transition from state A to state B, we search for a list of matches *M* between the effects in *E* and *B*. Effects match if they share the same DSP Algorithm. For effects in *M*, only the parameters of the DSP Algorithm and the audio mix settings for the effects are changed. Effects in *E* but not in *M* are faded out (or their ongoing fade-out is continued), and effects in *B* but not in *M* are instantiated, initialized, and faded in. When an effect in *B* has more than one match in *E*, preference can be given to effects using the following criteria:

1. prefer to match an effect with a similar input mix (it is better to change effect parameters than to reapply the effect to a new sound source);
2. prefer effects in the current state as opposed to effects from a previous state that are still fading out.

As an example (see Figure 2), suppose there is a state change from A to B. State A contains 2 delays and a filter effect, and state B contains a delay, a filter, and a ring modulator. The first delay and the filter in A would be reused as the delay and filter in B. The second filter in A would be faded out, and a new ring modulator would be created and faded in.

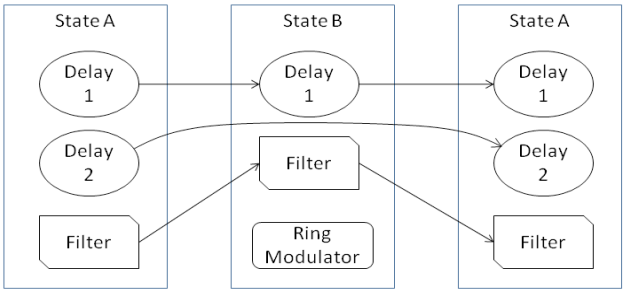

Figure 2. Active effects with matching DSP Algorithms are reused across state transitions.

Continuing the example, imagine that after the transition from A to B, there is an immediate transition back to A. The delay and filter, now part of state B, would be reassigned back to state A with their control parameters reset to values specified by state A. The second delay in A would be fading out after the transition to B. This delay would already have the correct parameter settings according to A, and it would be faded back in. The ring modulator does not match any of the effects in A, so its fade-in (in progress) is halted and replaced with a fade-out from the present level to zero.

4. USER INTERFACE

Once the architectural model is understood, the user interface is simple to understand and operate. It consists of several windows. The *input window* (see Figure 3) allows the user to type names (e.g. "violin") for input channels.

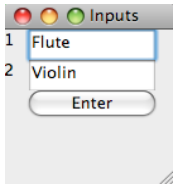

Figure 3. Input window names input channels.

The *state* window (see Figure 4) is used to create and edit states. The window has a drop-down menu to select a state for editing. When a state is selected, the controls in the state window are updated according to that particular state. (Provisions are also included to create new states, rename, and copy states.) The state window has multiple buttons or tabs to access effects within the current state. Each effect has a drop-down menu to select a DSP Algorithm, and when the algorithm is selected, the effect panel is populated with sliders and other controls to change parameters for that particular effect.

If an effect is active when parameters are adjusted, the parameters are immediately sent from the user interface to the corresponding audio object so that changes can be heard. It is also possible to edit states that are not active, in which case the new parameter values only take effect when the state (and its effects) become active.

The *Sequence* window (see Figure 5) is used to create sequences of states and mixer settings. Each element of the sequence is called a *scene*. Each scene denotes a state name to indicate what state should become active in the scene. Each scene also has mixer settings for mixing "dry" audio inputs to outputs. Finally, for convenience, there are settings for a global reverb effect that is always on but has adjustable levels and reverb time.

One of the problems of managing a sequence of scenes, each with many parameters, is that it becomes difficult to make global changes. One approach is to have global as well as local adjustments. For example, the gain applied to

input channel 1 could be the product of a "global" gain control and a "local" gain associated with the current scene. In AuraFX, we have taken a different approach where each control in a scene can be marked to "share" its value with the previous scene. By default, all values are shared among all scenes. Thus, changing the gain on input channel 1 in *any* scene changes the value in all scenes. To change the level at a particular scene, one first marks the control as "not shared" using the check box labelled "CHANGE," then adjusts the control. The change will affect only the current and future scenes, but not the previous ones. By setting all controls to "not shared," all controls are local and have no effect on other scenes.

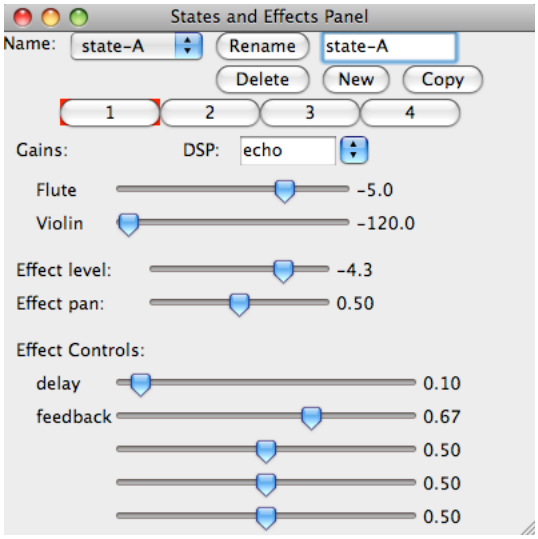

Figure 4. State window is used to edit states, each with a set of parameterized effects.

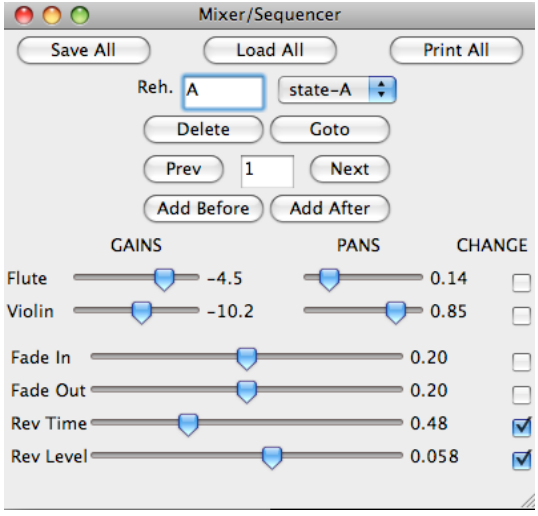

Figure 5. Sequencer window is used to create and arrange sequences of states, each with direct (dry) mix of inputs, fade-in and fade-out times, and global reverb.

It should be noted that each effect contains a "local" mix of inputs. These effect mixes will be reused each time the state appears in the sequence.

5. INTERACTION

In one mode of operation, the program advances linearly through scenes, applying the effects, and using specified fade-in and -out times at transitions. The system advances to the next state when a button is pushed on the graphical interface or in response to a MIDI or OSC [13] command.

For more interactive settings or rehearsals, the user can select a sequence position by name (names can be assigned corresponding to rehearsal markings in a score), by number, or through a MIDI program change command, or similar OSC command.

One limitation of this approach is all other performer interaction is only possible within the confines of an effect. An effect can analyze the audio input and respond to it in an interactive manner. Higher levels of interaction in which the actions of the performer select effects or change effect parameters are not supported. It is easy to imagine a more elaborate interface that maps MIDI control changes or OSC messages to effect parameters, and these could be controlled directly or indirectly by the performer.

6. EFFECT CREATION

One advantage of AuraFX over a dedicated multi-effects processor is that new DSP algorithms can be created and added to the system with minimal programming effort. A DSP algorithm can be implemented in several ways. First, the Serpent scripting language [14] can be used in a procedural manner to allocate and patch together unit generators. Second, the same approach can be taken more-or-less using C++, allowing efficient manipulation of samples, spectra, and other audio data in ways that might be difficult to implement using existing unit generators. Finally, a graphical patch editor [15] (see Figure 6) can be used to combine unit generators into an Aura "instrument" and output code in C++. Whether the C++ is generated by hand or by our patch editor, it must be compiled and a new AuraFX must be linked to incorporate the new code.

In addition to the DSP code itself, AuraFX needs a description of the DSP algorithm interface, including a name for display on the user interface, a list of parameter names, default values and ranges, and the name of a function that returns a new instance of the DSP algorithm as an Aura object. AuraFX uses a configuration file written in Serpent to build the necessary DSP algorithm descriptors. One could easily imagine an alternative approach using LADSPA or VST effects, which also have an API to obtain descriptions of their parameters.

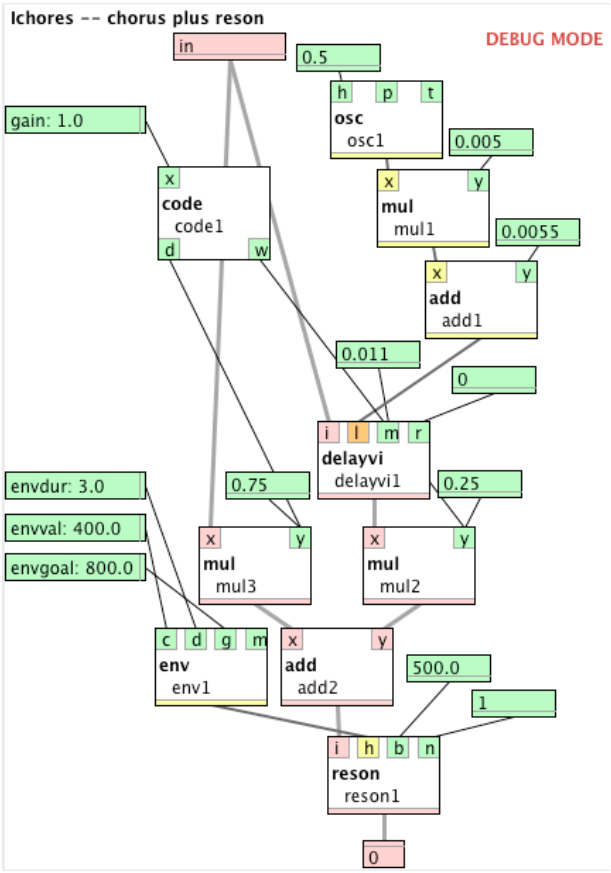

Figure 6. A graphical patch editor allows users to create new DSP algorithms for AuraFX effects.

7. DISCUSSION

AuraFX is a new open-source system that sits somewhere between a dedicated multi-effects processor and a programmable audio processing language system such as MAX MSP, Pd, SuperCollider, or Aura. Unlike conventional multi-effects processors, AuraFX includes important components for handling multiple input channels, simultaneous effects, controlled fade-in and fade-out times, mixer automation, and sequences of audio processing "scenes." In addition, AuraFX is extensible through the addition of new signal processing algorithms. Thus, AuraFX seems to be a more complete and flexible system for creating interactive audio compositions than a mere effects processor.

In contrast, programming-based music systems provide a more general and flexible environment for building interactive music systems. Unfortunately, this flexibility is usually obtained at the cost of building custom interfaces, mixers, and control strategies along with careful coding of transitions when effects change. While not as flexible as fully programmable systems, AuraFX does offer sophisticated effects-processing management that is difficult and therefore unlikely to be implemented for any single interactive composition.

The AuraFX architecture suggests some interesting directions for future computer music systems. Handling smooth transitions between states has been a long-standing and difficult programming problem. Another common problem is "backing up" in a linear structure during a rehearsal. These unplanned transitions are often untested and reveal bugs where variables are not reinitialized to proper values. The AuraFX architecture puts a layer of management between the control system (selecting states, adjusting parameters of effects) and DSP algorithms (actually processing sound). One benefit of this approach is that processing can be optimized, for example by turning off effects when their outputs fade to silence and by eliminating mixer channels when input gains are zero. These sorts of optimization are of course possible in general programmable systems, but they require careful programming and testing.

The down side of AuraFX is its limited control strategy. However, if one *does* need elaborate control schemes, it is likely that one will also need to enable/disable audio processing effects and manage smooth transitions between different configurations. Rather than throwing out this architecture and resorting to low-level code for smooth transitions and resource management, it might be much more productive to *retain* the AuraFX architecture and put the interactive controls at a higher level. The advantages would include (1) having a single, reusable implementation of some of the most difficult audio processing algorithms which manage unit generators, updating parameters, and making smooth fade-ins and fade-outs, and (2) having a higher-level control interface that operates in terms of states, effects, and parameters rather than direct communication with unit generators. From this perspective, AuraFX becomes a high-level audio processing runtime system. We hope to explore these possibilities in the future.

8. CONCLUSIONS

We have described an architecture for open-ended interactive computer music systems. While not as general as a programming language, we believe this architecture offers an interesting and useful compromise between ease of use and generality. The architecture also suggests that a high-level interface between control and signal processing aspects of programmable systems can simplify some difficult programming and resource management problems. In essence, if unit generators are the "assembly language" of computer music, we offer a higher-level language based on states and effects. Our implementation, AuraFX, has been used successfully in several performances. We hope this work will inspire others to think about new ways to build computer music systems that simplify the creation of new compositions.

9. REFERENCES

[1] Zicarelli, D. "An Extensible Real-Time Signal Processing Environment for Max," *Proceedings of the 1998 International Computer Music Conference.* San Francisco: International Computer Music Association (1998), pp. 463-466.

[2] Puckette, M. Pd. http://crca.ucsd.edu/~msp.

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

[4] Vercoe, B., "The Canonical CSound Reference Manual Version 5.07." Edited by J. ffitch, J. Piché, P. Nix, R. Boulanger, R. Ekman, D. Boothe, K. Conder, S. Yi, M. Gogins, A. Cabrera, F. Pinot, and A. Kozar. URL (accessed 26 Dec 2009): http://www.csounds.com/manual/html/index.html.

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

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

[7] Brad Garton, et. al. "RTcmix." http://music.columbia.edu/cmc/RTcmix/.

[8] Pope, S. T. and Ramakrishnan, C. "The Create Signal Library ('Sizzle'): Design, Issues and Applications." *Proceedings of the 2003 International Computer Music Conference*, San Francisco: International Computer Music Association (2003), pp. 415-422.

[9] Dannenberg and van de Lageweg, "A System Supporting Flexible Distributed Real-Time Music Processing," *Proceedings of the 2001 International Computer Music Conference*, San Francisco: International Computer Music Association (2001), pp. 267-270.

[10] Cook, P. and Scavone, G., "The Synthesis ToolKit (STK)," *Proceedings of the International Computer Music Conference,* International Computer Music Association, (1999), pp. 164-166.

[11] Place, T. & T. Lossius, "Jamoma: A modular standard for structuring patches in Max," *Proceedings of the 2006 International Computer Music Conference*. New Orleans, USA: International Computer Music Association (2006), pp. 143-146.

[12] Bencina, R. "Oasis Rose the Composition - Real-Time DSP with AudioMulch," *Proceedings of the Australasian Computer Music Conference*, ANU Canberra (1998), pp. 85-92.

[13] Wright, M., & Freed, A., "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers," *Proceedings of the 1997 International Computer Music Conference*. San Francisco: International Computer Music Association (1997), pp. 101-104.

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

[15] Dannenberg, R. "Aura II: Making Real-Time Systems Safe for Music," in *Proceedings of the 2004 Conference on New interfaces For Musical Expression*. M. J. Lyons, Ed. National University of Singapore, Singapore (2004), pp 132-137