A Flexible Real-Time Software Synthesis System[1]

Roger B. Dannenberg and Eli Brandt
Carnegie Mellon University
Pittsburgh, PA 15213 USA
`{dannenberg, eli}@cs.cmu.edu`

ABSTRACT: Aura is a new sound synthesis system designed for portability and flexibility. Aura is designed to be used with W, a real-time object system. W provides asynchronous, priority-based scheduling, supporting a mix of control, signal, and user interface processing. Important features of Aura are its design for efficient synthesis, dynamic instantiation, and synthesis reconfiguration.

1. Introduction

Software sound synthesis offers many benefits, including the flexibility to use a variety of algorithms, integration with control software and computer interfaces, and compact, portable hardware in the form of laptop computers. At present, software synthesis is limited by processor speed (and in many systems poor real-time operating system behavior and noisy audio interfaces). Faster machines, improvements in operating systems, and digital audio interfaces can solve all these problems. Announcements of various commercial software synthesis systems in the press indicate that we have moved from the realm of potential to reality.

We are interested in using software synthesis for real-time experimental music performance. To this end, we have designed, prototyped, and are implementing Aura, a complete software synthesis and control system. Our goal has several implications for our design, so we will describe some of our requirements before describing the design.

1.1. Requirements

Software portability is crucial to our work. We want to amortize our effort over at least a few generations of hardware and operating system changes. Systems such as the CMU Midi Toolkit and Csound illustrate the long life typical of comparable software systems. Furthermore, it is very uncertain what hardware/OS combination will deliver the performance we are looking for. Therefore, we must have the flexibility to run on different operating systems. In addition to raw computing speed required for sound synthesis, we are interested in systems that respond with low latency (requiring real-time support from the operating system) and systems that provide high performance computer graphics rendering for animation in multimedia performances. [Dannenberg 93] This requires hardware support and the availability of device drivers.

Flexibility is one of the main attractions of software synthesis, so our goal is not so much to build a specific synthesis engine (as in some commercial ventures) but to build a flexible platform or architecture that can be readily modified or extended to meet the needs of research and composition.

2. Design Decisions

High Level Languages lead to more readable, but sometimes less efficient code than assembler. Given the special nature of digital signal processing, we imagine that special-purpose code generators could also do a better job than general purpose compilers. However, in keeping with our requirements, we restrict ourselves to the use of a compiler (C++) to insure portability across different machine types. Without this portability, one could argue that a better approach would be to use DSP chips.

We use *floating point computation* throughout. On some current architectures, integer operations would be faster, but the trend is toward machines that are optimized for fast floating point computation. Also, considering the goal of flexibility, we believe that floating point is the only reasonable choice.

Dynamic Instantiation: software synthesis languages going back to Music V have created instances of instruments, but this can be a problem for real-time systems. Dynamic instantiation of new instruments means that the computation load can grow to exceed

[1]Published as: Dannenberg and Brandt, ‘‘A Flexible Real-Time Software Synthesis System,’’ in ***Proceedings of the 1996 International Computer Music Conference***, International Computer Music Association, (August 1996), pp 270-273.

the real-time capacity of the CPU. In our experience, dynamic instantiation is a very powerful mechanism worth having. Often, relatively simple mechanisms (similar to those used in commercial synthesizers) can be used to limit the number of instances.

Asynchronous control: One of the great promises of software synthesis is tight integration between synthesis and control, so we designed our system to support control as well as signal processing. Our experience with MIDI control systems [Dannenberg 93] indicates that control can take substantial amounts of computation, and this has important implications for the architecture. Software synthesis systems have traditionally used synchronous control in which control information is computed between each block of audio samples. The problem with this scheme is that long-running control computations can delay audio computation, causing buffer overflow and a corresponding pop on the output.

Although asynchronous software is more complex, it has the advantage that sound synthesis can proceed without waiting for control computation. If a control computation runs too long, it is preempted to compute sound. Prior to our experience with MIDI, we might have imagined that all computation should meet real-time constraints so synchronous control would be satisfactory. However, in our experience it is very convenient to consider control to be a more ‘‘soft’’ real-time task subject to occasional delays of many milliseconds. With MIDI, a delayed message does not normally cause a catastrophe because MIDI devices are asynchronously coupled to their control systems. Obtaining similar behavior in an all-software system requires architectural support.

3. Related Work

A number of commercial software synthesis systems have been implemented and/or announced in the trade press, but since the internal designs of these systems are proprietary, we cannot comment on this work here. Only a few systems have been described in the literature. Real-Time Csound [Vercoe 90] derives from the Music N languages. It uses synchronous control at the sample block rate and has uninterpolated control signals. The IMW software [Lindemann 91, Puckette 91] is more oriented toward complex interactive control. It also uses synchronous control at the sample block rate, but supports no dynamic instantiation. HTM [Freed 94] is a sound synthesis package for C programmers. It has been used in systems with asynchronous control running on multiple processors. Neither the IMW nor HTM have block-rate (i.e. control rate) signals per se, although presumably these can be added as synchronous block-rate control computations.

A problem with all of these systems is that their synthesis architectures make design choices that cost anywhere from 20 to 100% in computation time [Thompson 95]. In addition, we feel that a software synthesis system can offer better support for control, timing, and dynamic reconfiguration. Other relevant systems include Cmix [Lansky 87] (a non-real-time system) and Kyma [Scaletti 89] (which uses DSP chips for synthesis). While each of the systems mentioned offers some approach to our problems, none offers a very complete solution.

4. The Architecture

Our system is based on W [Dannenberg 95], a real-time programming environment that supports multiple zones of objects that communicate via messages. Message passing is synchronous within a zone and asynchronous between zones. In W, messages generally set object attributes. Normally, we expect all audio computation to take place within a single zone (see Figure 1). Within that zone, multiple sound objects (including familiar unit generators) are connected to perform the desired signal processing operations. We considered a purer functional programming model as in Nyquist [Dannenberg 92], but the functional programming model does not seem well suited to interactive control and flexible reconfiguration. Instead, we explicitly connect objects into a graph which then obeys functional, data-flow semantics.

Efficient signal communication is important. Given the use of W for other communication, W would be a logical choice except W is not designed for data-driven computation or shared memory communication. Therefore, we designed W with a simple mechanism for interconnecting sound objects, and we use W only to establish connections.

4.1. Interconnection

Signal processing computation in Aura is demand driven, and computation takes place a block at a time, where a block is some fixed number of contiguous samples. Our intended block size is approximately 32 samples, but this number can be changed easily. Any object that processes audio is called a sound object. A sound object has zero or more sound inputs, each denoted by a name (e.g. *Freq* or *Amp*) and zero or more sound outputs denoted by an integer index. Sound objects are derived from W objects so they may also receive W messages that set various internal parameters.

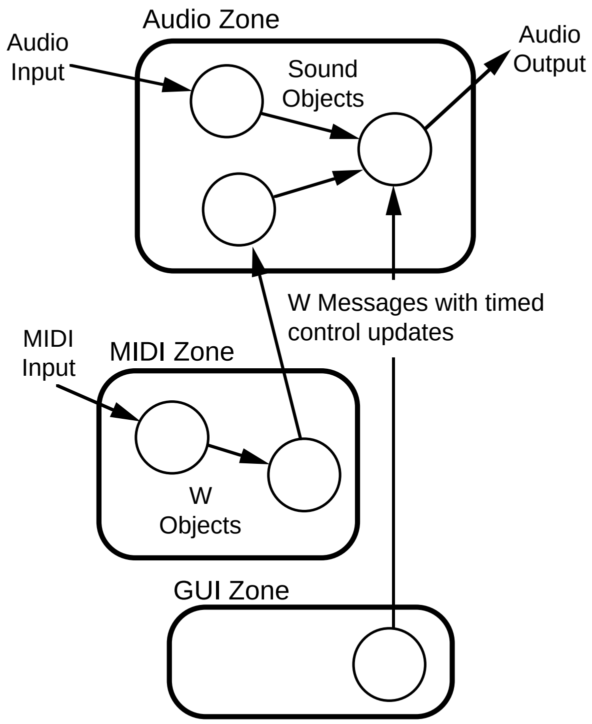

Figure 1: An Aura configuration with MIDI and Graphical User Interface zones providing asynchronous control.

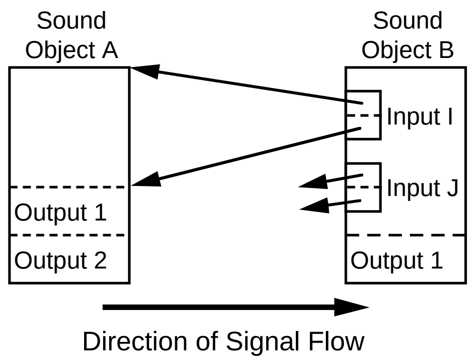

Figure 2: Sound objects and their interconnection

Figure 2 shows the connection of output 1 of object A to input I of object B. The connection is represented by two pointers in object B: one pointer is to object A and the other contains the address of the sample block through which samples are communicated. When B needs input samples, it first checks to see that A has computed samples for the current block (blocks are computed synchronously throughout the zone, so a zone-wide current block number is compared to a per-object counter). If A is up-to-date, the samples can be read directly from A's buffer. Otherwise, A is first called upon to compute the current block, and then computation continues. Notice that a call to compute the current block may recurse to other objects. In this way, an entire directed acyclic graph of interconnected objects is traversed for each output block.

This technique of checking each output for currency is equivalent to a topological sort on the graph of interconnected objects. We considered performing the sort explicitly and saving the resulting order of execution, but this would save only a small fraction of the execution time. Furthermore, the execution order needs to be recomputed at least every time a new connection is created.

4.2. Instantiation

Sound objects can be created dynamically to synthesize a new sound. A typical procedure would be to create a new sound object, initialize its fields, and connect it to some other object. All this can be done via W messages, e.g. in response to MIDI input.

A connection is made from the output of sound object A to the input labeled 'I' of sound object B by sending a W message of the form ''Set 'I' to A'' (the message is sent to object B). In the case where A has multiple outputs, the index of the desired output must be set in a previous message to B.

As a special case, the ''sum'' sound object class supports an arbitrary number of connected objects. For example, each instance of a note is attached to a sum object which outputs the sum of its inputs to audio output, reverb, or whatever. [Dannenberg 91]

4.3. Primitives

A key to efficiency is to build upon efficient signal processing primitives, where most of the processing takes place. We have performed extensive benchmarks in order to understand what factors are important in achieving efficient software sound synthesis. [Dannenberg 92] From our study, we learned that mixed sample rate computation is important, so we provide two sample rates: an audio rate and a control rate (corresponding to the sample block rate.) Additional sample rates can be supported so long as they are sub-multiples of the audio sample rate, allowing synchronous block boundaries.

Interpolation of control signals is another important consideration. Without interpolation, control rate envelopes can cause ''zipper'' noise unless the block size is very small, but small blocks have an adverse effect on performance. Overall, larger blocks (e.g. 32 samples) with linearly interpolated control signals seem to give the best performance. The cost of linear interpolation is more than offset by the savings of larger block sizes.

In our design, the primitive signal processing elements (i.e. unit generators) exist as C++ objects. Primitives have associated instance variables to hold parameters and to save state between sample block computations. A method is invoked to cause the primitive object to compute the next block of audio samples.

4.4. Structure

Sound objects are generally interesting only in combination, so any synthesis system must have a means for combining objects into larger structures such as instruments and orchestras. The structuring problem is especially interesting when dynamic instantiation is permitted, because the system then requires some internal representation for the structure of whatever will be instantiated.

One way to create a structured computation (i.e. instrument) is by writing C++ code to combine primitive objects analogous to instrument definition in Music V. The advantages of this structuring mechanism are (1) all input and output connections for an ‘‘instrument’’ are made to a single sound object, (2) at run-time, allocation is simple, fast, and atomic because initialization of sub-objects is handled by compiled code, (3) control-rate computation can be performed directly and efficiently by C++ code, and (4) communication among primitives within the sound object is handled by compiled code.

Alternatively, a single primitive can be ‘‘wrapped’’ with the appropriate sound object interface, allowing it to be connected to other sound objects. An ‘‘instrument’’ can then be constructed by interconnecting various primitive sound objects. This scheme has more overhead, especially when the instrument is instantiated, but it does have the advantage that new instruments can be built without recompilation. Aura supports both schemes.

4.5. Time

Time representation is the subject of much debate. Originally, the W system used millisecond timestamps in its messages, but for high sample rates and small block sizes, milliseconds may not have the precision to determine a unique block. Furthermore, some applications require that timestamps be precise to the sample or even sub-sample interval. [Eckel 95] Higher precision is a problem for 32-bit integers, though. A microsecond time unit will overflow 32 bits in a little over an hour. We decided to change W to use double-precision floating point numbers as timestamps. This gives very high precision and overflow protection, and the floating point format is easy to convert to other units such as sample or block counts. 64-bit integers would also be a good choice, but these are not supported by all compilers. Space prohibits a detailed discussion, but Aura implements block-synchronous updates by default. Mechanisms are in place to support down to sub-sample updates where needed.

4.6. Asynchronous Control

W allows a flexible combination of synchronous and asynchronous control. Within a zone, all objects execute non-preemptively. Synchronous control can be achieved by placing all control objects in the same zone as the audio synthesis objects. The computation of a block of samples will take place without preemption, but between block computations, all control operations will run to completion.

To achieve asynchronous control, control objects are placed in a separate lower-priority zone. Figure 1 shows MIDI and GUI zones providing control. Long-running control computations (e.g. redrawing a graphical slider) will then be preempted to allow computation of audio. Since communication between zones is by messages, and message delivery is always synchronous, updates are actually synchronous (this is usually a desirable feature).

In some cases, an atomic update of multiple parameters is necessary. Filter coefficients are an often mentioned case because filters can become unstable when updates are not synchronous. There are at least three mechanisms to achieve atomic updates. First, if a controller object is in the same zone as the controlled object, communication is synchronous, so multiple parameter changes can be sent without preemption. Second, W provides ‘‘multi-messages’’ which encapsulate a set of messages into one message that is delivered atomically across zones. Finally, using timed messages, updates can be sent for synchronous delivery at a specified time.

5. Summary and Conclusions

Aura is a new system for real-time software sound synthesis. It supports dynamic instantiation, asynchronous control, multiple sample rates and achieves this with greater efficiency than any other published architecture (based on benchmarks that compare different architectural approaches). To achieve flexibility, Aura is implemented as an extension of W, a distributed real-time object system that allows applications to be constructed by configuring components. W also enhances portability: With no changes to the DSP code, Aura can run as a process, a software interrupt handler, a device driver, or even a dedicated processor, using W to provide scheduling and communication in an implementation-independent manner.

References

[Dannenberg 91] Dannenberg, R. B., D. Rubine, T. Neuendorffer. The Resource-Instance Model of Music Representation. In B. Alphonse and B. Pennycook (editor), *ICMC Montreal 1991 Proceedings*, pages 428-432. International Computer Music Association, San Francisco, 1991.

[Dannenberg 92] Dannenberg, R. B. Real-Time Software Synthesis on Superscalar Architectures. In *Proceedings of the 1992 ICMC*, pages 174-177. International Computer Music Association, San Francisco, 1992.

[Dannenberg 93] Dannenberg, R. B. Software Support for Interactive Multimedia Performance. *Interface Journal of New Music Research* 22(3):213-228, August, 1993.

[Dannenberg 95] Dannenberg, R. B. and D. Rubine. Toward Modular, Portable, Real-Time Software. In *Proceedings of the 1995 International Computer Music Conference*, pages 65-72. International Computer Music Association, 1995.

[Eckel 95] Eckel, G., M. R. Iturbide. The Development of GiST, a Granular Synthesis Toolkit Based on an Extension of the FOF Generator. In *Proceedings of the 1995 International Computer Music Conference*, pages 296-302. International Computer Music Association, 1995.

[Freed 94] Freed, A. Codevelopment of User Interface, Control, and Digital Signal Processing with the HTM Environment. In *Proceedings of the International Conference on Signal Processing Applications and Technology*. 1994.

[Lansky 87] Lansky, P. *CMIX*. Princeton Univ., 1987.

[Lindemann 91] Lindemann, E., F. Dechelle, B. Smith, and M. Starkier. The Architecture of the IRCAM Musical Workstation. *Computer Music Journal* 15(3):41-49, Fall, 1991.

[Puckette 91] Puckette, M. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal* 15(3):68-77, Fall, 1991.

[Scaletti 89] Scaletti, C. The Kyma/Platypus Computer Music Workstation. *Computer Music Journal* 13(2):23-38, Summer, 1989.

[Thompson 95] Thompson, N. and R. B. Dannenberg. Optimizing Software Synthesis Performance. In *Proceedings of the 1995 International Computer Music Conference*, pages 235-6. International Computer Music Association, 1995.

[Vercoe 90] Vercoe, B. and D. Ellis. Real-Time CSOUND: Software Synthesis with Sensing and Control. In S. Arnold and G. Hair (editor), *ICMC Glasgow 1990 Proceedings*, pages 209-211. International Computer Music Association, 1990.