A Real-Time Audio Scheduler for Pentium PCs

Camille Goudeseune
Univ. of Illinois at Urbana-Champaign
603 W. Nevada #1
Urbana, IL 61801
cog@uiuc.edu

Michael Hamman
Univ. of Illinois at Urbana-Champaign
705 W. Nevada #4
Urbana, IL 61801
m-hamman@uiuc.edu

Abstract

The Audio Rendering Engine And Library (AREAL) offers a set of tools for developing real-time audio applications for Pentium computers running Windows 95/NT. It consists of a real-time scheduler and a framework for developing sound synthesis classes. AREAL lets composers and sound designers develop applications which address research into audio algorithms, their control in real time, and their correlation with non-audio processes. It emphasizes efficiency and low CPU overhead, coexisting well with other graphics and modeling computation possibly sharing its CPU. We briefly describe AREAL's design goals, architecture, and use.

1. Motivation

AREAL does software synthesis of audio in C/C++. It is intended more as a programmer's tool than a user's tool. It assumes no special (costly) hardware beyond current desktop computers, distinguishing it from DSP- and other hardware-based synthesizers. It is similar in intent to Aura and Nyquist [Dannenberg 96] but uses a more universal language than Lisp, and instead of striving for portability it runs on a platform with enough users to ensure longevity. Again, Max and its successor Pd [Puckette 97] emphasize connecting pre-existing objects and a graphical programming environment more than working at the C/C++ level. The feature sets of real-time Music-N languages, even those that encourage C programming, still reflect batch-mode use; AREAL has no such backwards-compatible ties. Finally, AREAL is well suited for audio applications where the limited timbral control and limited data bandwidth of MIDI hardware (and more recent software emulators) present difficulties [Moore 88]. Areal is small enough to be easily integrated into existing applications. It can use applications' own interfaces to networks, file I/O, and GUI instead of requiring its own. AREAL's small set of compositional assumptions lends itself to a wide variety of compositional techniques. For data sonification applications, AREAL allows for a "model-based" [Bargar 94] approach to audio in which auditory events are generated in real time as a direct mapping of a dynamically evolving data set or process.

2. Description

AREAL is a C-language API wrapped around a C++ library that handles real-time scheduling of audio samples to a standard sound card on a Pentium PC running Windows 95 or NT. AREAL incorporates (1) a real-time scheduler for managing uninterrupted playback of computed audio samples, and (2) an extendible sound synthesis library.

AREAL is targeted for composers and sound designers who wish to take a "roll-your-own" approach to computer music. Using AREAL, composers and audio researchers can develop audio software applications for sound synthesis and music composition, by adding synthesis libraries to AREAL and writing application code for implementing control interfaces. Composers and audio researchers desiring modest synthesis methods can build applications using synthesis objects already built into AREAL. These objects are initialized and controlled via the C API, an interface to code in AREAL's dynamically linked library (DLL). For composers who have developed synthesis algorithms which they would like to execute in a real-time application environment, AREAL provides a robust though easy-to-understand framework for folding that synthesis code into AREAL's real-time scheduler. The built-in synthesis algorithms then serve as models for incorporating these other algorithms into AREAL.

AREAL uses version 5.0 of Microsoft's Visual C++ compiler, which we chose over others for its good support tools, documentation, and speed of generated code. Even so, AREAL adds little overhead to the actual computation of audio data. Sample computation is done a block at a time ("vectorized", according to [Freed 92]); multi-

threaded execution avoids stalling the CPU; and the code executed for a parameter update is "thin", entailing only a range check, a table lookup, and an assignment operation. (AREAL offers a more elaborate mapping scheme from an application's state variables to the raw parameters of a synthesis algorithm, discussed in section 3.2.)

3. Software Architecture

AREAL has three nested software layers: the Scheduler, the Synthesis Engine, and a C-language wrapper. The Scheduler sends audio samples to the audio hardware, balancing low latency between control requests and audio results, safety from audio interruptions due to other processes needing the CPU, and scalable audio quality. The Synthesis Engine contains a hierarchy of sound synthesis classes and a facility for handling requests from the application and Scheduling layers. The wrapper API interfaces these two layers to the application.

The Scheduler requests audio streams from the Synthesis Engine and passes them on to the sound card. It requests samples from the Synthesis Engine to satisfy two constraints: avoiding interruptions in the audio stream and avoiding excess latency (due to excessive precomputing of samples). Computing samples some time before they are needed is a common strategy in real-time audio software systems: it allows samples from a particular synthesis algorithm to be computed a block at a time, reducing per-sample overhead (function calls, etc.) and improving use of the CPU and memory caches. This technique also reduces the likelihood of interruptions in the audio output on multitasking computers. The operating system may pre-emptively assign the CPU to internal tasks; if it does not provide the process which is computing audio data a consistently steady share of processor time, this audio process may become starved of CPU time and cause an audio interruption. Very short buffer times give low latency but often result in audio interruptions from actions as simple as moving the mouse. Longer buffer durations offer correspondingly greater resistance to audio interruptions but at the expense of possibly unacceptable latency. By default, AREAL uses a conservative 150 msec buffer, adequate for a 100 MHz Pentium running Windows 95. A computer running Windows NT or with a faster processor can use a shorter buffer, but if the computer is performing other tasks a near-ideal latency of under 5 msec remains unrealistic.

The Synthesis Engine contains a hierarchy of sound synthesis classes and a facility for handling messages from the Wrapper and Scheduling layers. Upon initialization, a list of sound objects is instantiated. When the Scheduler requests samples (through a callback function), the Synthesis Engine obtains buffers of samples from each synthesis object and sums them into the buffer that has been passed from the Scheduler.

Each class in AREAL's sound synthesis class library specifies a particular synthesis algorithm and defines a standard class interface. We envision an "open" sound synthesis class hierarchy, allowing extensions of current synthesis algorithms and addition of entirely new ones based on particular rendering and optimization requirements. The library is currently extensible at the source-code level. In a future version, however, we anticipate that this extensibility will be realized through implementation as an ActiveX container or through the Component Object Model paradigm. These Microsoft standards allow different applications to exchange information and for one application to embed itself into the other, in effect becoming a component of it. In this way, sound synthesis libraries could be "dropped into" AREAL.

The C-language wrapper acts as interface between these internals and an application. The interface exported by the AREAL DLL is purely a set of C functions. This simplifies the interface and lets applications written in languages other than C++ (notably Java) to use the AREAL DLL. The functions in this interface instantiate and control synthesis objects and also define data-to-renderer mappings.

3.1 Scalable Audio Quality

AREAL avoids sample underflow in a novel way by gracefully degrading audio quality. A facility called the OctaneMeister monitors the computer's CPU load. As this load nears 100%, the OctaneMeister tells each instantiated synthesis object to lower its computational "octane" requirement. Conversely, when the CPU load is low, synthesis objects gradually increase their level of octane use. How this is done is specific to each synthesis class. Moreover, synthesis objects are ranked in order of complexity: those whose performance would be most seriously compromised by lowering their octane use are placed at the bottom of the list.

Instead of holding synthesis parameters constant for each buffer of computation, AREAL synthesis objects can use a dynamically variable *k-rate* as described in [Freed 92]. One way synthesis algorithms can reduce CPU usage is by reducing the k-rates of various synthesis parameters, thereby reducing the frequency of recalculation of these

parameters. An extreme method computes samples at a fraction of the actual sampling rate, resulting in loss of high frequency content.

The OctaneMeister normally assumes that other processes sharing the CPU are *bounded* in their CPU use, *i.e.*, there is a fixed rate of execution of machine instructions which the other processes will not exceed no matter how much "idle time" is available, and this fixed rate is in fact less than the maximum rate which the CPU can support. The OctaneMeister has a second mode of operation for working with processes that are not CPU-bounded, such as certain 3D video games which acquire as much CPU as they can in order to maximize their visual frame rate. In this mode, the application itself is responsible for allocating resources to its components (audio, graphics, modelling) and ensuring that none of them starve. So rather than adapting to CPU load, the OctaneMeister follows the application's explicit commands for how much CPU time to give to the audio synthesis objects. In this scenario, the default "polite" behavior from the OctaneMeister would fail in the face of "greedy" behavior from the other processes, and audio would end up running at its lowest possible quality.

3.2 Mapping Application Data to Audio Parameters

It is often desirable to control the parameters of a synthesis algorithm in ways other than individually and separately. To this end, AREAL implements the separate concepts of *model parameters* and *synthesis parameters*. A model parameter belongs to the application sending data to AREAL, for example the position or speed of an object or the rate of execution of some process in the model. A set of synthesis parameters, on the other hand, is precisely the run-time interface to a synthesis algorithm, consisting of scalars such as carrier-to-modulator ratios or filter coefficients. In a particular application, each synthesis parameter might be a function of one, several, or no model parameters; a complete set of these functions is called a *mapping*. An example illustrates the syntax for assigning several synthesis parameters s_j from model parameters m_i:

```
arealSetMapping(hSynthObj, "s1=(m1,10,40); s2=.3*(m1,10,40)+4*(m2,.5,-.5); s3=0");
```

Here s_1 varies from zero to one as m_1 varies from 10 to 40; s_2 is a weighted sum of two similarly normalized values, the first as before, the second varying from 0 to 1 as m_2 varies from 0.5 down to -0.5; finally, s_3 is a constant. (The identifier *hSynthObj* is a handle to a particular synthesis object.) This mapping is defined once at initialization; during execution, it is evaluated lazily whenever a model parameter changes. Such changes are communicated to areal with function calls like

```
arealSetParm(hSynthObj, "m1", 39);
    arealSetParmAll("m1", 39);
```

where the first of these functions updates a single synthesis object, and the second automatically updates all synthesis objects whose mappings use the model parameter m_1.

4. Using AREAL

The function *arealInitAudioLib*() initializes the AREAL subsystem with a particular sampling rate and number of audio channels. Computation of audio starts and stops with calls to *arealStartAudio*() and *arealEndAudio*() respectively. Synthesis objects of a particular synthesis class are instantiated and commence computing audio samples with *arealAddSynth*(); this function returns a handle to the new synthesis object. The functions *arealSetMapping*(), *arealSetParm*(), and *arealSetParmAll*() control running synthesis objects as described above. Finally there are functions which control the OctaneMeister's behavior: *arealSetCPULimit*() sets a limit on AREAL's CPU usage; *arealSetCPUMode*() and *arealSetCPUValue*() are intended for applications with nonbounded CPU usage.

This example of an AREAL synthesizer is a sample-and-hold noise generator . Class *SynthParm* encapsulates the smooth updating and k-rate functionality, used here for amplitude and hold time.

```
class SynthNoise: public Synth {
   SynthParm ampl;    // amplitude scalar
   SynthParm seconds; // how long to hold this value of the rnd. num gen.
   float duration;    // how many seconds until the next rand() call
   Samp sampValue;    // current output value (floating point)
   void AllBgn(void)       { ampl.Bgn(); seconds.Bgn(); }
   void AllUpdate(float z) { ampl.Update(z); seconds.Update(z); }
   void AllUpdateLast()    { ampl.UpdateLast(); seconds.UpdateLast(); }
```

```
    void AllEnd(void)       { ampl.End(); seconds.End(); }
};

SynthNoise::SynthNoise() {
    ampl.Reset(.5); seconds.Reset(.001); duration = 0.f; sampValue = 0;
}

void SynthNoise::SetSparm(const char* synthparamName, float z) {
    if (!strcmp(synthparamName, "holdtime")) seconds.Set(z);
    if (!strcmp(synthparamName, "ampl"    )) ampl.Set(z);
}

void SynthNoise::Compute(Samp* rgsamp, int csamp, int dsamp) {
    for (int i=0; i<csamp-dsamp; i+=dsamp) {
        if ((i & 255) == 0)
            AllUpdate(i/(float)csamp);    // k-rate is every 256 samples
        ComputeSample(rgsamp+i, dsamp);
        }
    AllUpdateLast();
    ComputeSample(rgsamp+i, dsamp);
}

inline void SynthNoise::ComputeSample(Samp* psamp, int dsamp) {
    duration += arealSRinv;
    if (duration >= seconds.Get()) {    // it's time to compute another value
        sampValue = ((rand() & 0xffff) - 32768) * ampl.Get();
        duration -= seconds.Get();
        }
    *psamp = sampValue;
    DoStereo(psamp, dsamp);
    }

main() {
    arealInitAudioLib(44100, 1);         // mono, 44K
    h1 = arealAddSynth(arealSynthNoise);
    arealSetMapping(h, "holdtime = .01 + .01*(x, 0, 1); ampl = (y, 0, 1)");
    arealStartAudio();
    Sleep(1000);                  // listen to default sound for 1000 msec
    arealSetParm(h, "x", .7);
    Sleep(1000);                  // listen to a hold time of .017
    arealEndAudio();
}
```

The full source code for AREAL is available at `www.shout.net/~mhamman/Areal/`.

5. Conclusions

AREAL is currently being used for the kernel of the Windows port of NCSA's VSS sound server [Bargar 94], as well as the synthesis layer of the composition environment Orpheus [Hamman 97]. These two applications illustrate two approaches to using AREAL. Orpheus has rich control structure but relatively straightforward synthesis algorithms; these are implemented directly as AREAL synthesis objects. VSS, on the other hand, already had a large library of synthesis algorithms which had to work and sound the same as the SGI version; this entire library is treated as a single "synthesis algorithm" by AREAL, and components of VSS use their own control mechanisms instead of availing themselves of AREAL's individual *arealSetParm*() synthesis controls.

6. References

[Bargar 94] R. Bargar, I. Choi, S. Das, C. Goudeseune, "Model-Based Interactive Sound for an Immersive Virtual Environment." *Proc. ICMC*, 1994. San Francisco: International Computer Music Association. pp. 471-474. (VSS is available at `www.ncsa.uiuc.edu/~audio` .)

[Dannenberg 96] R. Dannenberg, "A Flexible Real-Time Software Synthesis System." *Proc. ICMC*, 1996. San Francisco: ICMA. pp. 270-273.

[Freed 92] A. Freed, "New Tools for Rapid Prototyping of Musical Sound Synthesis Algorithms and Control Strategies." *Proc. ICMC*, 1992. San Francisco: ICMA. pp. 178-181.

[Hamman 97] M. Hamman. "Composition of Data and Process Models: a Paralogical Approach to Human / Computer Interaction." *Proc. ICMC*, 1997. San Francisco: ICMA. pp. 109-112.

[Moore 88] F. R. Moore, “The Dysfunctions of MIDI.” *Computer Music Journal* 12(1): 19-28. 1988.

[Puckette 97] M. Puckette, “Pure Data.” *Proc. ICMC*, 1997. San Francisco: ICMA. pp. 224-227.