The CREATE Signal Library ("Sizzle"): Design, Issues, and Applications

Stephen Travis Pope and Chandrasekhar Ramakrishnan
Center for Research in Electronic Art Technology (CREATE)
University of California, Santa Barbara (UCSB)
email: {stp, sekhar}@create.ucsb.edu

Abstract

The CREATE Signal Library (CSL) is a portable general-purpose software framework for sound synthesis and digital audio signal processing. It is implemented as a C++ class library to be used as a stand-alone synthesis server, or embedded as a library into other programs. This first section of this paper describes the overall design of CSL version 3 and gives a series of progressive code examples. We also present CSL's facilities for network I/O of control and sample streams, and the development and deployment of distributed CSL systems. What is more interesting is the discussion that follows of the design issues we faced in implementing CSL, and the presentation of a few of the applications in which we've used CSL over the last year.

1 Introduction

This document describes the CREATE Signal Library (CSL, pronounced "sizzle"), a flexible, portable, and scalable software framework for sound synthesis and digital signal processing. The following sections describe the basic system requirements and present the design and its implementation of version 3, with extensive code examples along the way.

The initial design of CSL dates back to 1998 (it was then called the CREATE Oscillator, or CO), but the current incarnation was started by students in the MAT 240D *Sound Synthesis Techniques* course at UCSB in the Spring of 2002. A C++ implementation of a minimal sound synthesis framework (in less than 1000 lines) was developed by the first author and introduced at the start of the class, and during the quarter the students added a large number of synthesis classes (refining the basic framework significantly as they went).

In the year since that time, CSL has continued to evolve as we used it for several larger applications (desribed below), and a revised version of the core framework—version 3—was written (primarily by the second author) in the Spring of 2003. During these reimplementations, a series of design discussions were held, which corresponded with later graduate courses in the MAT 240X series, especially a course on programming interfaces (APIs) for real-time sound streaming, and the most recent course on spatial and surround sound programming.

CSL is now an open source project; the current source code and documentation can be retrieved over the Internet from the CREATE Web site at http://create.ucsb.edu/CSL.

1.1 What CSL is

CSL is a simple yet powerful library of sound synthesis and signal processing functions. It is packaged as an object-oriented C++ class hierarchy for standard DSP and computer music techniques, and is suitable for integration into existing applications, or use as a stand-alone synthesis/processing server.

CSL is similar to the JSyn (Burke 1998), CommonLispMusic (Schottstaedt 2000), STK (Cook and Scavone 2002), and Cmix (Pope 1993) frameworks in that it is packaged as a library in a general-purpose programming language, rather than being a separate "sound compiler" as in the Music-N family of languages (Pope 1993). We have already used CSL to build stand-alone applications, interactive installations, MIDI instruments, and light-weight plug-ins for DSP tools.

CSL is designed from the ground up to be used in distributed systems, with several CSL programs running as servers on a local-area network, streaming control commands and sample buffers between them. We describe these facilities in more detail below.

1.2 What CSL is not

CSL is not a music-specific programming language such as Music-N or SuperCollider (McCartney 1996); rather, CSL programs are written in standard C++ and then linked with the CSL library. CSL has no graphical user interface (as in Max/Pd [Puckette 1996] or Kyma [Scaletti 1989]), but it is expected that GUIs will be built that manipulate "patches" and "scores" for CSL. CSL is not a music representation language such as Smoke/Siren (Pope 2001), rather it is a low-level synthesis and processing engine. (We use CSL to build synthesis engines that can be controlled from Siren applications via net-

work or MIDI messages.) CSL has no scheduler, it simply responds to in-coming control messages as fast as it can.

This flexibility means, however, that CSL can serve a number of different purposes, from being used as a plug-in library for other applications to serving as the basis of synthesis servers for other front-end languages, such as MPEG4/SAOL.

1.3 Design Goals

The composers and researchers at CREATE need a scalable, portable, and flexible network-driven sound synthesis package. "Scalable" means that the system needs to be able to support what we call "orchestral-scale" sound synthesis—large groups of instruments with complex synthesis models and dynamic multi-modal control, mixed and spatialized out to 16 or more output channels. This scalability will be achieved by running clusters of CSL-based synthesis and processing server programs on many computers connected by a fast local area network. "Portable" means that the software must not depend on a particular hardware platform or operating system. CSL is written entirely in "generic" C++ and uses hardware abstraction classes for I/O ports, network interfaces, and thread APIs. "Flexible" means that the library should support several techniques of software sound synthesis, digital audio signal processing of sound files or live input, and also be appropriate for use as a signal processing library for embedding into other applications. "Network-driven" is important because we plan to separate user input and control gesture mapping onto different computers than those performing the actual sound synthesis and spatialization.

From the start, we decided that CSL had to run on Linux, UNIX (Solaris, IRIX, OpenBSD), and MacOSX; MS-Windows is supported as well, though some features of CSL (primarily the networking support and multi-threaded processing) are missing on that platform. We require it to support all popular sound synthesis and processing techniques, and it must be callable over a local-area network via the OSC (Freed and Wright 1997) or CORBA (http://www.omg.org) protocols. It should send its output samples either directly to an output device, or to a network socket (e.g., connected to a remote spatialize/mix/play program). For scalability, multiple CSL processes running on different machines had to support inter-machine sample streaming and be integrated into the CREATE Real-time Application Manager (CRAM, Pope *et al*. 2001) distributed application development/deployment framework.

Given these basic requirements, a whole range of design issues arise in the process of implementing such a software framework. We will present the current CSL system, and discuss the design solutions as they come up.

1.4 A Quick Example

Within a CSL program, there are C++ objects that correspond to what are called "unit generators" in traditional software sound synthesis languages—sound sources, processors, mathematical operations, etc. These can be connected together using C++ variables to represent the inputs and outputs of the unit generator objects.

As an initial example, consider a sine wave oscillator to which an amplitude envelope is applied. The CSL C++ code for this is shown below. (Comments are preceded by "//" in C++.)

```
// Create a sine wave oscillator named "vox"
// with a frequency of 220Hz.
   Sin vox(220.0);
// Create an ADSR envelope named "env";
// the arguments to the constructor are
// (duration, attack, decay, sustain, release).
   ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
// Create a signal multiplier named "mul" giving it
// the oscillator and the envelope as its inputs.
   MulOp mul(vox, env);
// Set the multiplier as the client of the output driver
   io.set_root(mul);
```

To run this example, one needs to "include" the main CSL header file in the source code file, call the C++ compiler with the source, and link the resulting object code file with the CSL class library. We will discuss how CSL handles the program's "main" function later.

When this example program executes, it creates the unit generator objects—the oscillator, the envelope generator, and the multiplier—and then tells the output driver (the global variable *io*) that its "root" output object is the multiplier. The output driver then periodically requests buffers of samples from the multiplier.

When this happens, the multiplier asks each of its inputs for a buffer of data and multiplies them. We call this the "pull model" of synthesis; each time the output object requests a new buffer of samples, the "tree" of CSL unit generator objects is traversed with each object requesting sample data from its inputs.

As an aside to demonstrate the flexibility of CSL objects, note that we used the envelope object in the preceding example as if it were an "envelope generator" on an analog synthesizer, and the multiplier as a kind of "voltage-controlled amplifier." CSL envelopes can also be used as "processors," in that they

can scale a dynamic input, allowing us the re-write the example as follows.

```
// Simplified sine-with-envelope example using
// the envelope object as a combined gernerator
// and amplifier
    Sin vox(220.0);
    ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
    env.set_input(sin);
    io.set_root(env);
```

1.5 Components

The CSL library and default "main" program consists of several components:

- the object framework for the synthesis/processing engine;
- the unit generator class library;
- the start-up, configuration, and system save/restore facilities;
- the OSC control interfaces;
- the database interface for sound samples and spectra; and
- the CRAM "Service" interface for management of multiple CSL instances on a network.

2 Inside The CSL Framework

An instance of a CSL-based program is characterized by its graph of DSP units, generally a number of "patches" (subgraphs) connected to a mixer object as in other software sound synthesis programs. In the simplest case, the DSP graph can be a single unit generator, e.g., a fixed-waveform oscillator connected directly to the output. A CSL DSP graph has a single "root" node, usually the output unit generator or a mixer that takes several subgraphs as its inputs. Envelope and instrument objects allow subgraphs to be triggered independent of one another, and define the notion of active versus turned-off subgraphs for increasing the efficiency of mixers.

Each instance of a CSL-based program can implement multiple voices, possibly using different synthesis techniques. CSL instances are dynamically reconfigurable, though we have avoided dynamic DSP graphs on the applications we've built to date.

2.1 The Core CSL Framework

CSL is based on an object-oriented domain model that consists of abstractions for:

- objects that create or process blocks of sound samples (Buffer, FrameStream, SampleStream, Processor, etc.);
- objects representing control variables (StaticVariable, DynamicVariable);
- objects that connect to I/O drivers (IO and its subclasses); and
- objects that help manage CSL "patches" and instrument libraries (Instrument).

The evaluation of the DSP graph is triggered by the "pull" of an IO object (an instance of a subclass of IO), which is typically connected to a direct output API such as PortAudio (Bencina and Burke 2001), CoreAudio, to a socket-based network protocol, or to a sound file. The IO object holds onto the "root" of the DSP graph, and periodically calls the root's *next_buffer()* function, passing it a pair of sample buffer objects (input and output).

In the basic CSL framework, there is no essential difference between constant values, control signals, and audio signals. DSP graphs can also incorporate unit generators running at different sample rates, default buffer sizes, and number of channels, so control-rate generators (and parallel expansion of multichannel processing) are possible.

2.2 The Synthesis/DSP Classes

The heart of CSL is its unit generator and signal processing class library: the subclasses of FrameStream. There are several flavors of signal and control sources including wavetable oscillators (in both perfect and band-limited versions), noise sources, chaotic generators, FFT/IFFT, and others.

Signal processors such as filters and panners are objects that take signal synthesis graphs as their inputs and manipulate the sample buffers their inputs generate. These are all subclasses of both FrameStream and the mix-in class Processor. CSL includes canonical-form and FIR filters, panners, mixers, convolution, and flexible delay lines.

Simple operators such as addition and multiplication of signals are handled by the AddOp and MulOp unit generators.

Sampled sound files can be loaded using several sound file formats, and SoundFile objects can play them back into a DSP graph.

Envelopes are handled as breakpoint functions of time. Breakpoints can occur in the middle of a sample buffer, and the envelope class handles the subsegments properly. There are helper classes that provide constructor methods for the standard envelope types: Triangle, AR, ADSR, various windows, etc.

Plugging unit generators together is simple, one can simply use the output of one as an input, e.g., to the *set_frequency()* function, of another (see the examples below). To scale and offset dynamic control functions, "variable" objects are provided by the CSL framework.

2.3 CSL Mixer/Spatializer Programs

CSL instances can have their own direct output

objects (to a sound output interface on the local machine), or they can send their output (blocks of samples) through sockets to another mixer/reverberator/spatializer/play program. We have designed a protocol based on the UDP network interface in which data packets have a header that incorporates an instance ID and sequence number. CSL servers can then run on multiple machines in a server farm that have no special audio IO hardware.

The mixer and spatializers are, in fact, simply CSL-based programs that perform no actual synthesis, but rather read sample blocks from other CSL instances (over a network) and process them.

To complement this brief introduction to the CSL framework, we will move on to a series of code examples. A discussion of the details of the implementation will then follow.

3 Code Examples

These annotated code examples are intended to give the reader a taste of CSL programming;a much more complete set of examples is included in the CSL manual; see http://create.ucsb.edu/CSL. As mentioned above, patch editors can and have been built that will allow non-C++-literate users to construct and combine CSL DSP graphs.

3.1 Simple Oscillators and Patching

```
// Create a 220 Hz sine-wave oscillator object
// named "vox" using the Sin class.

// Sin class constructor function
        Sin vox(220);
// Plug it in to the global output driver (io).
        io.set_root(vox);

// Use a 3 Hz. sine to amplitude-modulate a
// sine wave in a multiplier.

// Create two oscillators, one with an assigned
// frequency and one with the default.
        Sin vox(220), mod;
// Set the frequency of the second oscillator.
        mod.set_frequency(3);
// Multiply (amplitude modulate) the two.
        MulOp mul(vox, mod);
        io.set_root(mul);
```

3.2 Processing and Filtering

```
// Using a sine wave for L/R panning.
        Sin vox(220), pos(2);   // signal, panner
// A panner takes an input and a position function.
        Panner pan(vox, pos);
        io.set_root(pan);

// Apply a band-pass filter (300 - 700 Hz
//  [= 500 +- 200]) to pink noise.
        PinkNoise pnoise (20000);
        ButterworthFilter filter(pnoise, pnoise.rate(),
                        kFilterBandPass, // type
                        500, 200);          // cf, bw
        io.set_root(filter);
```

3.3 Spectral Processing

```
// Create a spectrum with odd harmonics and
// perform inverse FFT synthesis.
// Create an IFFT oscillator.
        IFFT vox;
// Set some data in the spectrum
// (freq, amplitude, phase).
        vox.set_partial(1, 0.5, 0);
        vox.set_partial(3, 0.25, 0);
        vox.set_partial(5, 0.05, 0);
        vox.set_partial(9, 0.01, 0);
        io.set_root(vox);
```

4 The Implementation of CSL

CSL is written in portable C++, with plug-in synthesis modules written as subclasses of an abstract unit generator class. The primary class hierarchies are described in the following sections. CSL uses 32-bit floating-point numbers to represent samples (though this can be changed with a single definition to allow for integer or higher-precision floating-point processing). All processing is done in blocks, which are typically between 32 and 1024 sample frames in size.

4.1 The FrameStream Class Hierarchy

The main CSL declarations are in the file FrameStream.h, which defines the following classes:

- Buffer, the basic n-channel sample buffer class;
- FrameStream, the frame stream class, the central abstraction to CSL3;
- SampleStream, a 1-channel frame stream;
- Processor, a mix-in for framestreams that process an input frame stream;
- Writeable, a mix-in for framestreams that one can write into;
- Phased, a mix-in for framestreams with phase accumulators;
- Positionable, a mix-in for framestreams that one can position; and
- IO, an input/output stream or driver abstraction.

Instances of the Buffer class represent multi-channel sample buffers; they have memory pointers to sample storage (which may be placed in a special heap or pool) as well as a set of flags about the storage state (allocated, zero, populated, etc.). The class has methods to allocate, zero, and free sample stor-

age, and several convenience methods.

FrameStreams represent objects that can generate buffers of frames. ("Frame" refers to a collection of samples that are designed to be played [or manipulated] simultaneously.) This class is the root of all functions and unit generators. The key methods FrameStreams implement are:

- *next_buffer()* - make a buffer's worth of frames
- *next_value()* - answer just one value (sample)
- *is_fixed_over()* - say if my value is fixed in the next buffer

The actual function signature of the *next_buffer()* method is,

```
// get a buffer of samples
// this is the core CSL "pull" function
virtual status next_buffer(Buffer & inputBuffer,
                           Buffer & outputBuffer);
```

Note that an input buffer is provided; it represents the (optional) input sample buffer coming from the A/D convertors. The return value is a status flag—a member of a special enumeration—which we use (rather than exception handling) throughout CSL.

Since buffers are inherently multichannel, but many standard computer music unit generators are not, the default behavior of *next_buffer()* is to call a monophonic version of itself (called *mono_next_buffer()*) for each output channel. Subclasses of FrameStream are free to override this with different behaviors, allowing true mono unit generators, copy-mono-to-all-output-channels, or various other multi-channel behaviors.

The function *is_fixed_over()* is used for some optimizations where we know that the FrameStream will only generate one value over the next *n* frames.

SampleStream is a FrameStream of special importance; it is a one-channel frame stream that calls the monophonic *next_buffer()* method and then copies the single-channel buffer data to all output channels (using *memcpy()*). The default unit generators, Operators, and Variables are all SampleStreams. This is not a necessity, but it is a convenience: it makes the internals of CSL much simpler.

4.2 Envelopes

Control functions are often most-simply described as break-point envelope functions. Breakpoints can occure in the middle of buffers. Interpolation between breakpoints can be linear, exponential, cubic, or use other interpolation algorithms.

4.3 Kernel Helper Classes

The class Gestalt has class (static) methods for the sample rate, default buffer size, safe memory allocation, etc.

One useful subclass of FrameStream that bears special mention here is the ThreadedFrameStream, which uses a background thread to compute samples. It caches some number of buffers from its "producer" sub-graph and supplies them to its "consumer" thread immediately on demand. It controls the scheduling of the thread of its producer. While this obviously introduces latency within a DSP graph. It is a known latency with no latency jitter.

Interleaver is a helper class for taking non-interleaved sample buffers (as used within CSL and by the Apple CoreAudio API), where the samples for each channel are stored in a separate array, and copying them into and out of interleaved sample buffers (as used by several common I/O APIs, including PortAudio).

To accomodate FrameStreams and Processors that might want to have different buffer sizes, a BlockResizer object can be placed between two elements of a DSP graph. This buffers calls to its up-stream client into groups of a different block size than the *next_buffer()* calls it receives. The main application of these is in graphs that use time-frequency transforms, so that, for example, one can use a wavelet (or Fourier) transform with a large window size in a graph that needs to be run with low I/O latency.

4.4 Variables

Variable objects permit CSL programmers to use constants anywhere signals are expected, and to do simple scaling (multiplication) or offset (addition) of dynamic signals and constants.

4.5 The IO Classes

All activity within a CSL program is triggered by some output object calling the *next_buffer()* function of some FrameStream. The simplest IO object is an interface to a sound output device driver that receives call-backs from the operating system at a regular rate (the sample rate divided by the output buffer size), and forwards them to the root of its DSP graph. Other IO classes are available that write samples to sound files, or receive output requests via a network socket and pass their data packets back over the same socket (these are called UDP_IO ports, see below).

Note the importance of this for real-time performance; input control commands come in asynchronously (e.g., via OSC or MIDI), and the synthesis process is driven by output calls coming from another thread of control. Thus CSL has no internal notion of time, but unit generators may have state, e.g., related to their current phase or indices within envelope control functions. Thus, the minimal granularity of tim-

ing is the IO buffer frame rate, e.g., ~1 msec for 64-frame output blocks and a sample rate of 44 kHz.

4.6 RemoteFrameStreams and UDP_IO

A RemoteFrameStream is a FrameStream that is connected by a UDP network socket to another CSL process. In response to the *next_buffer()* call, the RemoteFrameStream sends a UDP request to its server to get the next sample buffer. The server is assumed to be on a remote machine, and is a CSL program that uses a UDP_IO object as its output "driver." The request packet sent to the server causes the server to call its DSP graph's *next_buffer()* method and return the sample buffer to the client via a UDP message.

To set this up, the server must be a CSL program, and the UDP_IO object must know what port it listens to. The client (the RemoteFrameStream) needs to know the server's host name, the port it listens on, and the port that the client is to listen on for response packets. The client first sends the server an "introduction" packet with its IP/port so that the server can open a response socket. Then the client can send the server sample buffer requests.

4.7 Instrument Classes

There are several utility classes to make it easier to manage DSP graphs. A Instrument object has a DSP graph, a set of reflective accessors, and a list of envelopes. The DSP graph is the instrument's "patch," the accessors describe what the control parameters of the patch are (i.e., their names, types, and "setter" functions), and the envelope list is the collection of envelopes that need to be triggered to start a new note.

With this abstraction of a graph, one can easily construct code that automatically creates the mapping "glue" to control CSL programs from OSC or MIDI. As an example, a simple instrument might create several accessors in a list with the following code.

```
list[0] = new Accessor("du", set_duration_f,
                CSL_FLOAT_TYPE);
list[1] = new Accessor("am", set_amplitude_f,
                CSL_FLOAT_TYPE);
```

A special start-up method can take a "library" (a list of Instrument objects) and generate an OSC address space like the following.

```
/i1/          instrument 1 (simple example)
    /i1/du:   set-duration command
    /i1/am:   set-amplitude command
```

4.8 Input and Control

Using the instrument/accessor framework, one can set up CSL programs to respond to commands coming in from a variety of sources, such as OSC, MIDI, CORBA messages or from score file readers.

5 Using CSL

There are several ways to compile CSL programs, and several versions of the *main()* function to be used for CSL programs. For many kinds of applications, CSL is not even involved in the *main()* function. Since CSL is simply a C++ class library, one can easily reuse it in any number of ways:

- incorporate it as a component of another application (e.g., a game);
- use CSL to build plug-ins, e.g., for Steinberg's VSL API or Apple's CoreAudio API; or
- build an application with a graphical user interface that controls CSL synthesis and processing.

The generic CSL *main()* function is used for testing, and calls an arbitrary test function that can be supplied by the user. This function generally sets up a DSP graph (the test to be run), plays a note, and then exits.

Another configuration uses a file reader that parses and executes a score file in an abstract ASCII version of MIDI (described elsewhere).

The most common interactive version of CSL uses a *main()* function that sets up an OSC address space (given an instrument library as an array of CSL instrument objects, see above) and waits for in-coming OSC messages to set control values and trigger instrument envelopes.

As we mentioned above, CSL is designed from the ground up to be used in distributed systems, with several CSL programs running as servers on a local-area network. The companion paper in these Proceedings discusses the distributed processing framework in more detail.

6 Open Design and Implementation Issues

As we mentioned in the Introduction, the design of a framework such as CSL forces one to make a number of policy decisions that have various impacts on subsequent implementation details. Over the generations of evolution of CSL, we have sought novel solutions to a number of age-old design issues in computer music software.

We should admit here that both authors of this paper are more accustomed to programming in Smalltalk than in C++, and that several important design decisions were made by the coin-toss method (with later evaluation of the impact of the coin toss).

Some of our issues relate to building portable, robust, and fast C++ software. These are require-

ments that contradict each other to some extent.

For example, the C++ standard template library provides vector and list classes, but these are both quite slow (relative to using C-style arrays and pointers), and not entirely portable to the platforms we care about.

In general, CSL classes are conservative about buffer allocation. No allocations are ever done at runtime, and unit generators (even processors) try to reuse the buffers they are given whenever possible.

The CREATE Oscillator (which was essentially CSL version 0), used object pointers throughout; while we have now moved to using references in function calls, there are still problems in several places with reference counting.

We still occasionally have reason to debate the relative merits of interleaved (as in PortAudio) vs. non-interleaved (as in CoreAudio) sample storage. The main impact is seen in processors such as filters

We have also tried several different approaches to representing buffers of data in other domains (e.g., FFT or wavelet spectra), in the cases that they're even exposed.

The handling of *is_fixed_over()*, *is_linear_over()*, and the facilities for flexible (multi-rate) control-rate processing will be revisited in the next revision.

The last interesting open question is whether or when FrameStreams should know their assumed number of channels; there are cases where this might be very important, and others where it's better to make some reasonable default assumption.

7 Applications

Since the Winter of 2002, we have used CSL for several very different applications. We introduce these next.

7.1 Sensing/Speaking Space

Sensing/Speaking Space is an interactive audio/video installation developed by one of us (Pope) in collaboration with the media artist George Legrady; it premiered at the San Francisco Museum of Modern Art in February, 2002. In the installation, a computer vision system analyzes the movement of spectators in the gallery and sends OSC messages to a sound synthesis server. The first version of the sound server was written in SuperCollider (version 2), but suffered from persistent reliability problems (intermittent crashing), an excessive memory foot-print (1 GB), and poor debuggability (no SuperCollider debugger).Starting in January, 2003, *Sensing/Speaking Space* was rewritten in C++ using CSL.

While a detailed evaluation of the re-write and in-depth comparison of CSL and Supercollider is beyond the scope of this document, the new version of *Sensing/Speaking Space* premiered in a gallery performance in April of 2003, ran very reliably for a week, and sounded just like the first version. In both cases, the source code for the piece totals about 1200 lines, includes several helper classes, and incorporates a simple GUI with sliders to mix the various layers. The performance was also comparable between the SuperCollider versions in that a 500 MHz Apple G4 PowerBook was kept quite busy running the synthesis and spatialization engine with 6 output channels (and overtaxed to produce 8 output channels).

Given the reasons why it was necessary to port *Sensing/Speaking Space* from SuperCollider to C++, the CSL framework stood up quite well to its first real-world performance.

7.2 Ouroboros and OndeCorner

Ouroboros is an application for processing, sampling, and looping audio input and sound files. In this case, CSL is not used for the processing. Ouroboros hosts AudioUnits, the standard plug-in format on MacOS X, and lets the user create graphs of AudioUnits for adding effects to sound. Ouroboros employs CSL to simplify the reading and writing of sound files and for capture and looping of audio.

OndeCorner is an AudioUnit plug-in built using CSL. OndeCorner transforms sound to the wavelet domain and lets the user modify wavelet coefficients with a variety of processes. The resulting wavelet coefficients are inverse transformed to the time domain to product the output.

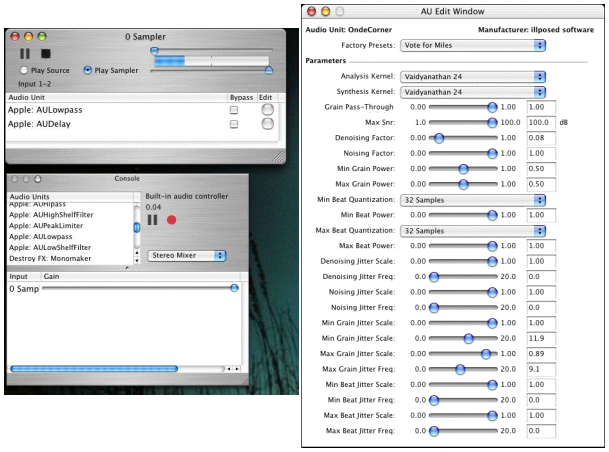

Figure 1: Ouroboros and OndeCorner Screens

In addition to being an example of a plug-in written in CSL, OndeCorner is a showcase for using CSL to integrate DSP code from other sources. CSL can be used to easily take code that was not designed specifically for processing audio, and apply it to audio

domain processing. In this case, we used the Wave++ (http://www.scs.ryerson.ca/~lkolasa/CppWavelets.html) from Ryerson Polytechnic University for a wavelet transform implementation. After building a simple wrapper class, we could use apply their wavelet transform to real-time audio signal processing.

7.3 Reverb Plug-in

In a recent UCSB graduate course on spatial sound, students developed a series of panners, reverberators, and spatializers based on the CSL framework. Later, CSL was used for a convolution-based reverberator and HRTF-based spatializer that uses the FFTW library for the analysis and synthesis.

7.4 The Expert Mastering Assistant (EMA)

Our largest current project using CSL is an expert system that uses fine-grained multi-level music analysis to suggest parameters for signal processing to be applied during music mastering. We're using a combination of CSL, AudioUnits, and third-party DSP code along with multi-dimensional scaling functions and a blackboard system for application management. The figure on the right illustrates the current (July, 2003) mock-up of the EMA user interfaces, showing the output and logging pane on the left. The central pane is the metering and control pane, with several kinds of signal displays and a transport control. The right-most pane is for control of the real-time mastering signal processing.

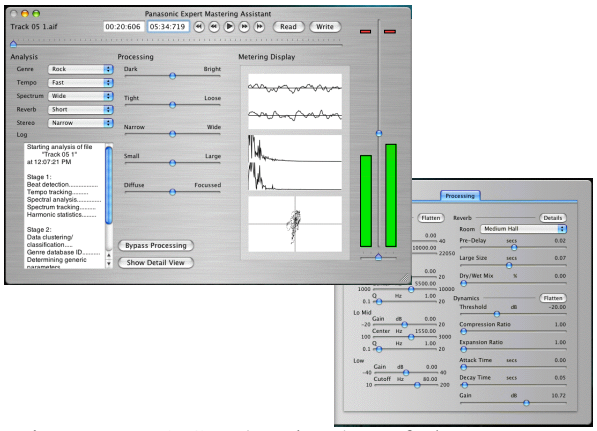

Figure 2: EMA GUI (mock-up) as of 7/2003

8 Plans for Future Work

There are several enhancements underway at present in the CSL workgroup. Some are related to adding new synthesis methods such as various kinds of physical and spectral modeling, while others relate to providing better integration between CSL and the CREATE CRAM System infrastructure.

9 Conclusion

The CREATE Signal Library is a working, open-source, portable, flexible sound synthesis and processing engine. The CSL class library can be used to construct stand-alone synthesis/processing servers, or can be integrated into other applications that require some sound generation or processing functions (e.g., games, music software, web-based services, or educational applications). The complete CSL manual is on-line at http://create.ucsb.edu/CSL.

10 References

Burk, Phil. 1998. "JSyn–A Real-time Synthesis API for Java." *Proc. 1998 ICMC.*

Bencina, Ross and Phil Burk. 2001. "PortAudio: an Open Source Cross Platform Audio API." *Proc. 1998 ICMC.*

Cook, Perry R. and Gary P. Scavone. 2003. "STK software documentation." http://www-ccrma.stanford.edu/software/stk.

Freed, Adrian and Matthew Wright. 1997. "Open Sound-Control: A New Protocol for Communicating with Sound Synthesizers." *Proc. ICMC 1997.*

McCartney, James. 1996. "SuperCollider: a new real time synthesis language." *Proc. 1998 ICMC.*

Pope, S. T. 1993. "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal* 17(2).

Pope, S. T. 2001. "Music and Sound Processing in Squeak Using Siren." in Guzdial, Mark and Kim Rose. *Squeak: Open Personal Computing and Multimedia.* (book and CD-ROM) Prentice-Hall.

Pope, S. T., A. Engberg, F. Holm, and A.Wolf. 2001. "The Distributed Processing Environment for High-Performance Distributed Multimedia Applications." in *Proc. 2001 IEEE Multimedia Technology and Applications Conf.*, U. C. Irvine.

Puckette, Miller. 1996. "Pure Data." *Proc. 1996 ICMC.*

Pope, S. T. and C. Ramakrishnan, 2003. "Recent Developments in Siren: Modeling, Control, and Interaction for Large-scale Distributed Music Software." *Proc. 2003 ICMC.*

Ramakrishnan, C. 2003 *Musical Effects in the Wavelet Domain*. Graduate Thesis: Media Arts and Technology Program, UCSB.

Sandell, Greg. 1998. *SHARC: The Sandell Harmonic Archive.* see http://www.parmly.luc.edu/parmly/sharc.html

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13:(2). reprinted in S. T. Pope, ed. 1991. *The Well-Tempered Object.* Cambridge, Massachusetts: MIT Press.

Schottstaedt, W. 2000. *Common Lisp Music Documentation*. see http://ccrma-www.stanford.edu/software/clm.