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 general-purpose software framework for sound synthesis and digital audio signal processing. It is implemented as a C++ class library to be used to build stand-alone synthesis servers, or embedded into other programs. This paper describes the overall design and implementation of CSL version 3. We also present CSL's facilities for network I/O of control and sample streams, and the development and deployment of distributed systems. What is more interesting is the discussion that follows of the design issues we faced in CSL, and the presentation of several 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 software framework for sound synthesis and digital signal processing. The initial design of CSL dates back to 1998 (it was then called the “CREATE Oscillator,” or CO); the current incarnation was developed with students in the MAT 240D *Digital Audio Programming: Sound Synthesis Techniques* course at UCSB in the Spring of 2002. A C++ implementation of a minimal real-time sound synthesis framework (in less than 1000 lines) was developed by the first author and introduced at the start of the class; during the quarter the students added synthesis classes, refining the framework significantly as they went.

In the year since that time, CSL has continued to evolve as we used it for several applications (described below), and a revised version of the framework (CSL3) was written (primarily by the second author) in the Spring of 2003. During this period, a series of design discussions were held, which corresponded with later graduate courses in the MAT 240X series, especially a course on programming interfaces for real-time sound streaming, and the most recent on spatial and surround sound software.

CSL is now an open source project; the current source code, examples, 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.

Similar to JSyn (Burke 1998), CommonLispMusic (Schottstaedt 2000), STK (Cook and Scavone 2002), and Cmix (Pope 1993), CSL is packaged as a class/function library in a general-purpose programming language, rather than being a stand-alone “sound compiler” as in the Music-N family of languages (Pope 1993). This flexibility means, however, that CSL can serve a number of different purposes, including the development of stand-alone synthesis programs, serving as a plug-in library for other applications, or supporting programs written in scripting languages or in MPEG4/SAOL format.

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, Pope and Ramakrishnan 2003), 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 network or MIDI messages.) CSL has no scheduler, it simply responds to in-coming control messages as fast as it can; using small block sizes allows one to minimize the system latency.

1.3 Design Goals

In designing CSL we tried to balance several, sometimes conflicting, goals. We envisioned a system that would execute efficiently, and also be straightforward to extend. We wanted the system to be transparent—statements should be understandable and have clear resource costs (minimizing the behind-the-scenes "magic"). At the same time, we wanted the library to be flexible and easy to use, and we wanted the framework to be platform-independent and scalable. These goals placed certain real constraints on our design.

The design of the CASL kernel—the frame stream hierarchy—attempts to encourage efficient code, while keeping the model as simple as possible. We tried to provide facilities for implementing optimizations, but made use of these facilities optional, so that it is possible to get new frame streams up and running with minimal effort.

We tried to implement objects in a way that they do only what they say they do, and do not go out of the way to do extra things for you. This makes easier to estimate, by inspection, the CPU and memory costs for a CSL DSP graph. As an example, the library has support for running different subgraphs at different block sizes (useful for spectral transformations), but this does not happen automatically. The user has to put a block resizer object at the appropriate point in the DSP graph.

For portability, we tried to restrict ourselves to very generic C++. We do use the Standard Template Library (STL), though, which presents portability issues on Microsoft platforms. When appropriate, we use abstraction classes for platform specific APIs such as audio I/O, network interfaces, and threads.

When considering scalability, we aimed for 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 channels. To that end, we accept control data via OSC (Freed and Wright 1997), and have I/O objects that stream samples over network sockets, allowing CSL programs to be distributed across multiple machines. Multi-server configurations are managed by the CREATE Real-time Application Manager (*CRAM*, Pope et al. 2001).

After introducing CSL in more depth, we will return to reconsider some design issues.

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 input and output ports.

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.
    Sine 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 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 the results. 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 or control 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," 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 with fewer unit generators as follows.

```
// Simplified sine-with-envelope example using
// the envelope as a processor
    Sine vox(220.0);
    ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
    env.set_input(sin);
    io.set_root(env);
```

The third option is to use the oscillator's "scale" input (essentially a "voltage control"), allowing us to write the example as,

```
// Sine-with-envelope example using the sine's
// scale (volume control or AM) input
    Sine vox(220.0);
    ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
    vox.set_scale(env);
    io.set_root(sin);
```

2 Inside CSL

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 complex graphs.

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.

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.

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, UnitGenerator, Processor, Phased, etc.);
- objects representing flexible control variables (StaticVariable, DynamicVariable);
- models of standard software sound synthesis "unit generator" DSP modules;
- objects that connect to I/O drivers (IO and its subclasses, socket-based stream objects); 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), Apple's CoreAudio, to a socket-based network protocol, or to a sound file.

In the 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, buffer sizes, and number of channels, so control-rate generators (and parallel expansion of multichannel processing) are possible.

2.2 The FrameStream Class Hierarchy

The CSL kernel consists of the following classes:

- **Buffer**, the basic n-channel sample buffer class;
- **FrameStream**, the frame stream class, the central abstraction of CSL;
- **SampleStream**, a 1-channel frame stream;
- **UnitGenerator**, adds scale and offset controls;
- **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;
- **Cacheable**, a mix-in for classes that may or may not cache signal buffers (e.g., of a wavetable);
- **Positionable**, a mix-in for frame streams that one can seek on; and
- **IO**, an input/output stream or driver abstraction.

Instances of the Buffer class represent sample buffers; they know their number of channels and buffer size, and have memory pointers to sample storage (which may be managed individually or 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 storage, 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()* - fill in a buffer's worth of frames;
- *next_value()* - answer just one sample value; and
- *is_fixed_over()* - answer whether the receiver's value is fixed for a given range of samples.

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

```
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.

The number of frames in the output buffer determines the block size of the *next_buffer()* call; it may change between calls (e.g., for latency control).

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, allowing true mono unit generators, copy-mono-to-all-output-channels, or various other multichannel behaviors.

The function *is_fixed_over()* is used for some optimizations where we know that the FrameStream will generate a fixed value over a buffer of frames.

SampleStream is a FrameStream of special importance; it is a one-channel frame stream that uses the monophonic *next_buffer()* method and then copies the single-channel buffer data to all output channels. The default unit generators, operators, and variables are all SampleStreams. Fancier multichannel processors can inherit directly from FrameStream.

2.3 The Synthesis/DSP Classes

CSL incorporates several flavors of signal and control sources including sum-of-sines and wavetable oscillators (in both ideal and band-limited versions), noise sources, chaotic generators, FFT/IFFT, sound file readers, 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 subclasses of both **FrameStream** and the mix-in class **Processor**. CSL includes canonical-form and FIR filters, panners, mixers, convolution, and flexible delay lines.

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. There are helper classes that provide constructor methods for the standard envelope types: Triangle, AR, ADSR, various windows, etc. Envelopes can act as processors (as in the second code example above), and any signal (e.g., a wavetable or sound file) can be used as a control function.

2.4 Operators and Variables

Simple operators such as addition and multiplication of signals are handled by the **AddOp** and **MulOp** unit generators (subclasses of **BinaryOp**).

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.

2.5 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).

In 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 (the output driver's callback thread). 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 timing is the IO buffer frame rate, approx. 1.4 msec (per layer of OS sample buffering) for 64-frame output blocks and a sample rate of 44 kHz. One can tune the latency of a CSL program dynamically by varying the call-back block size.

2.6 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.

2.7 Kernel Helper Classes

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

A **ThreadedFrameStream** 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 (ideally) no latency jitter.

An **Interleaver** takes 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 accommodate FrameStreams and Processors that use 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.

2.8 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, CORBA, 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
```

2.9 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.

3 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.

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 (Pope and Ramakrishnan 2003) discusses the distributed processing framework in more detail.

4 Code Examples

A few more code examples should give the reader a better feel for CSL programming; a much more complete set of examples is included in the CSL manual (see http://create.ucsb.edu/CSL).

4.1 Simple Oscillators and Patching

```
// Use a 3 Hz sine to amplitude-modulate (scale)
// a 220 Hz sine wave.
        Sin vox(220), mod(3);
        vox.set_scale(mod);
        io.set_root(vox);
```

4.2 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);
```

4.3 Spectral Processing

```
// Create a spectrum with odd harmonics and
// perform inverse FFT synthesis.
        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);
        io.set_root(vox);
```

5 Open Design Issues

As mentioned in the earlier section on design, we have had to make trade-offs between efficiency, simplicity, transparency, and ease-of-use. Throughout the evolution of CSL we have sought solutions to age-old design issues in computer music software in a way that was appropriate with how we were using CSL at that moment. There remain capabilities that we have considered, but have not yet had a compelling need to implement. And there are some choices that we have made the we find ourselves continually revisiting. (The first version of CO was designed for maximum simplicity, and proved indeed to be “good enough to complain about.”)

The highest-level design issue in a library such as this is the overall orientation of the object model: is the central abstraction a signal, a buffer, a stream, an operation, or what? This and several other choices were motivated by the I/O API we were driving with CSL (generally PortAudio or CoreAudio). For example, the structure of the Buffer was partially motivated by the desire to have a minimal “impedance mismatch” with the I/O, and partially by the desire to simplify the most common case where processing of multi-channel data happens independently on each channel. This is an area where we have continuing debates. Eventually, we would like to unify the handling of Buffers with the handling spectral and wavelet data, but have not yet figured out a satisfactory solution.

Other choices were motivated by the problem domain of real-time digital signal processing. Software engineering principals generally favor the use of exceptions over status flags, but we chose status flags nonetheless because they have a more transparent run-time costs. We occasionally find ourselves reconsidering this choice.

The C++ standard template library provides abstract collection classes such as vector and list, but these are quite slow relative to using C-style arrays and pointers. We use vectors sparingly in the CSL core, e.g., the buffer object has a vector of sample pointers for the multichannel sample data.

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

CSL has always been used as a tool for pedagogy and experimentation. This has led us to err on the side of simplicity over optimal efficiency. We want it to be easy to write a new frame stream from scratch and hear the results as quickly as possible. Because of that, we don't have much fancy handling of control rate frame streams, nor do we optimize graphs for cache utilization. We do see the possibility that the need to use CSL in “prime-time” software changing this balance, but we hope we can find solutions that improve efficiency without complicating the implementation of frame streams.

Previous incarnations of CSL passed objects by pointer, but in CSL3 we switched to passing objects by reference. We felt that this gave the library a more “C++” flavor, however, we readily confess that while we are both comfortable with C/C++, our object-oriented heritage is really Smalltalk, not C++ (a caveat that should be applied to this whole endeavor). Thus we may have made decisions that seemed natural to us but may not to a “native” C++ programmer.

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.

Some things we have considered implementing, but have not yet been compelled to do include: overloading arithmetic operators on frame streams, SIMD (AltiVec) implementations of frame streams, and an abstraction for a graph of CSL frame streams.

6 Applications

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

6.1 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. 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). In early 2003, *Sensing/Speaking Space* was rewritten in C++ using CSL.

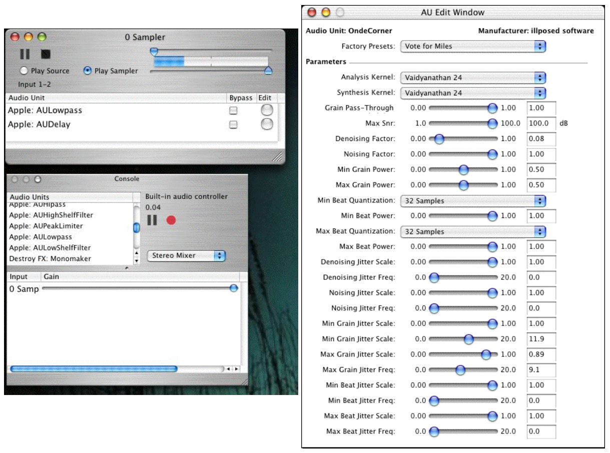
While a detailed evaluation of the re-write and in-depth comparison of CSL and Supercollider is beyond the scope of this document, the new version of *Sensing/Speaking Space* premiered in a gallery performance in April of 2003, ran very reliably for a week, and sounded just like the original version. In both cases, the source code specific to 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 two versions in that a 500 MHz Apple G4 PowerBook was able to run 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.

6.2 Ouroboros and OndeCorner

Ouroboros is an application for processing, sampling, and looping audio input and sound files built by the second author. 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 (Ramakrishnan 2003). The resulting wavelet coefficients are inverse transformed to the time domain to produce the output.



Audio Unit: OndeCorner   Manufacturer: illposed software

Factory Presets: Vote for Miles

Parameters

Analysis Kernel:	Vaidyanathan 24			
Synthesis Kernel:	Vaidyanathan 24			
Grain Pass-Through	0.00	1.00	1.00	
Max Snr:	1.0	100.0	100.0	dB
Denoising Factor:	0.00	1.00	0.08	
Noising Factor:	0.00	1.00	1.00	
Min Grain Power:	0.00	1.00	0.50	
Max Grain Power:	0.00	1.00	0.50	
Min Beat Quantization:	32 Samples			
Min Beat Power:	0.00	1.00	1.00	
Max Beat Quantization:	32 Samples			
Max Beat Power:	0.00	1.00	1.00	
Denoising Jitter Scale:	0.00	1.00	1.00	
Denoising Jitter Freq:	0.0	20.0	0.0	
Noising Jitter Scale:	0.00	1.00	1.00	
Noising Jitter Freq:	0.0	20.0	0.0	
Min Grain Jitter Scale:	0.00	1.00	1.00	
Min Grain Jitter Scale:	0.0	20.0	11.9	
Max Grain Jitter Scale:	0.00	1.00	0.89	
Max Grain Jitter Freq:	0.0	20.0	9.1	
Min Beat Jitter Scale:	0.00	1.00	1.00	
Min Beat Jitter Freq:	0.0	20.0	0.0	
Max Beat Jitter Scale:	0.00	1.00	1.00	
Max Beat Jitter Freq:	0.0	20.0	0.0	

Figure 1: Ouroboros and OndeCorner Screens

In addition to being an example of a plug-in written in CSL, OndeCorner uses 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 as the wavelet transform implementation. After building a simple "wrapper" class, we could use their wavelet transform for our real-time audio signal processing.

6.3 Panner and Reverb Plug-ins

In a recent UCSB graduate course on spatial sound, students developed a series of surround-sound up-mixers, multichannel panners, reverberators, and spatializers based on the CSL framework. Later, CSL was used for a convolution-based reverberator and HRTF-based spatializer.

6.4 Lua Patcher

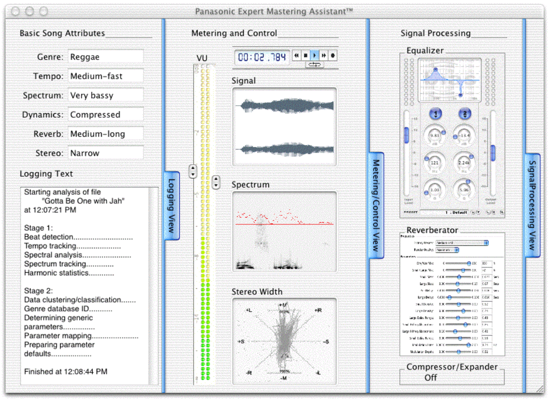
Since the first designs, we have planned to use a scripting language as a front-end for CSL. The motivations were to have a rapid-turn-around development environment for CSL DSP graphs. After looking into several options (python, TCL, lua, Ruby, FScript, etc.), we chose lua (http://www.lua.org) because it is built to be "embeddable" (i.e., into other applications), it is very simple, and it compiles to a virtual machine language for run-time speed (rather than being interpreted). The down-sides are that its object-oriented facilities are not as complete as, for example, python, and that it is not as well known as some of the other options.

As an example, the following lua function creates a Lorenz chaotic oscillator and pans it in stereo.

```
-- Lua program for a panning chaotic oscillator
test_panning_chaos = function ()
    lorenz = new(Lorenz);
    envargs = {0.5, 0.0, 0.0, 0.003, 0.5, 0.5, 0.0};
    envelope = new(Envelope, envargs);
    panner = new(Panner, {lorenz, envelope});
    audioout(panner);
end
```

6.5 The Expert Mastering Assistant

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 below illustrates the current (July, 2003) mock-up of the EMA user interface, 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.



Genre: Reggae
Tempo: Medium-fast
Spectrum: Very bassy
Dynamics: Compressed
Reverb: Medium-long
Stereo: Narrow

```
Starting analysis of file
        "Gotta Be One with Jah"
at 12:07:21 PM

Stage 1:
Beat detection............................
Tempo tracking.....................
Spectral analysis.......................
Spectrum tracking.............
Harmonic statistics............

Stage 2:
Data clustering/classification.......
Genre database ID...........
Determining generic
parameters...............
Parameter mapping.....................
Preparing parameter
defaults..................

Finished at 12:08:44 PM
```

Compressor/Expander
Off

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

7 Plans for Future Work

There are several enhancements underway at present in the CSL workgroup. Some are related to adding new synthesis methods—physical and spectral modeling or granular synthesis—while others relate to providing better integration between CSL and the CRAM distributed processing system infrastructure.

We also intend to do in-depth profiling of CSL programs to look for optimization, and to write some of the core functions using the AltiVec SIMD features of the IBM/Apple G5 processor.

8 Conclusion

The CREATE Signal Library is a working, open-source, portable, flexible sound synthesis 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 source code and manual can be found at http://create.ucsb.edu/CSL.

9 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.