Center for Research in Electronic Art Technology
University of California, Santa Barbara
http://www.create.ucsb.edu

The CREATE Signal Library (“Sizzle”): An Introduction

Stephen Travis Pope and Chandrasekhar Ramakrishnan

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 document describes the overall design of CSL version 3 and gives a series of progressive code examples.

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 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, and a revised version of the core framework was written (primarily by Chandrasekhar Ramakrishnan) in the Spring of 2003.

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.

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), CommonLispMusic (Schottstaedt), STK (Cook), and Cmix (Lansky) 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). 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. These CSL DSP servers receive control commands via the network and send their output sample blocks to other servers over the network. (Technically put, all control in CSL is transmitted via Open Sound Control [OSC] network messages and any “plug” between elements in a CSL DSP graph can be deferred over a network socket.) A typical large-scale CSL configuration is illustrated in Figure 1 below.

In this example, each of the round-edged rectangles is a separate server program. The top four servers are CSL programs; the larger box in the middle is the CREATE Distributed Processing Environment (DPE, [Pope and Engberg]) manager, and the server at the bottom captures and maps user input (e.g., from instrumental performers).

The control links (shown as dotted lines) use the Common Object Request Broker Architecture (CORBA [OMG]) and Open Sound Control (OSC [Wright and Freed]) protocols, and the inter-program sample streams (drawn as arrows) use the CSL sample block protocol. The “Yellow System” distributed application framework manages the distributed CSL application, starts up the individual CSL servers, and routes control messages coming from the gestural input devices to the appropriate server.

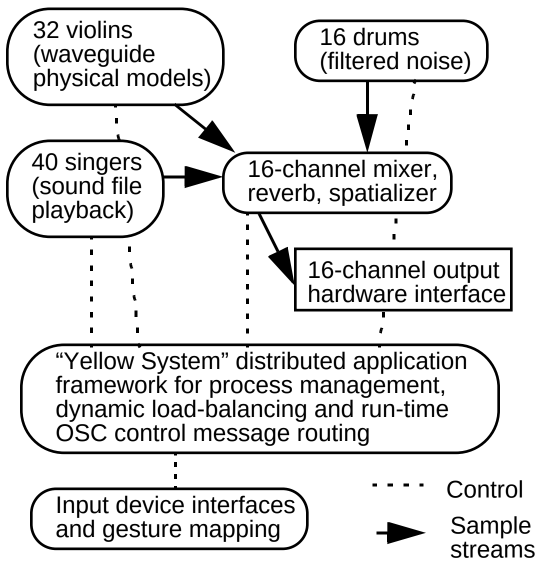
32 violins (waveguide physical models)
16 drums (filtered noise)
40 singers (sound file playback)
16-channel mixer, reverb, spatializer
16-channel output hardware interface
“Yellow System” distributed application framework for process management, dynamic load-balancing and run-time OSC control message routing
Input device interfaces and gesture mapping
Control
Sample streams

Figure 1: A distributed CSL configuration

What CSL is not

CSL is not a music-specific programming language such as Music-N or SuperCollider (McCartney); rather, CSL programs (i.e., CSL-based servers) are written in standard C++ and then linked with the CSL library. CSL has no graphical user interface (as in Max [Puckette] or Kyma [Scaletti]), 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 (Pope), rather it is a low-level synthesis and processing engine. CSL has no scheduler, it simply responds to in-coming control messages (received, e.g., via OSC) 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.

Design Goals

Users 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 control, mixed and spatialized out to 16 or more output channels. This scalability will be achieved by running CSL server programs on many computers connected by a fast local area network (as illustrated in Figure 1 above). “Portable” means that the software must not depend on a particular hardware platform or software operating system. CSL is written entirely in “generic” C++ and uses hardware abstraction classes for I/O ports and network interfaces. “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) 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 or CORBA 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 “Yellow System” distributed application framework.

A Quick Example

Within a CSL program, there are C++ software objects that correspond to what are called “unit generators” in traditional software sound synthesis., e.g., sound sources, processors, mathematical operations, etc. These can be connected together using C++ variables to represent the unit generator objects.

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

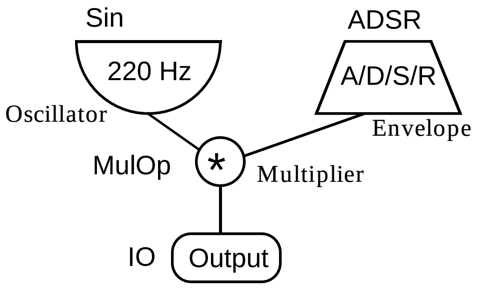

Figure 2: A simple CSL "patch"

```
// 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 global
// output driver "io."
    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 below.

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.

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 Yellow System interface for management of multiple CSL instances on a network.

We will discuss each of these in the sections that follow.

Inside The CSL Framework

An instance of the CSL program is characterized by its graph of DSP units, generally a number of "patches" 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.

Each instance of the CSL program can implement multiple voices, possibly using different synthesis techniques. CSL instances are dynamically reconfigurable.

The Basic 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
- objects that help manage CSL "patches"

and instrument libraries (Instrument).

The evaluation of the DSP graph is driven by calls from the output mechanism asking for buffers of samples. An IO object (an instance of a subclass of IO) is typically connected to a direct output API such as PortAudio (Burke, Bencina, et al.), CoreAudio, or to a socket-based network protocol.

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) Each buffer object knows its number of channels, number of sample frames, and has internal storage for the actual sample data. In the case of a multi-unit DSP graph "patch," the root unit will pass the *next_buffer()* call "up" the graph, e.g., an oscillator asking its dynamic frequency input for its next buffer of control information before doing its computation.

In the basic CSL framework, there is no essential difference between constant values, control signals, and audio signals. Unit generator objects can, however, ask their inputs whether they are constant over a given range of samples using the *is_constant_over(number_of_samples)* message. If a unit generator answers "true" to this call, the client can get a single value from it using the *next_sample()* message instead of *next_buffer()*. DSP graphs can also incorporate unit generators running at different sample rates and default buffer sizes, so control-rate generators are possible.

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, and others.

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

Simple operators such as addition and multiplication of signals are handled by the AddOp and MulOp unit generators. Their inputs can be signals, constants, or variables that take signals or controls and can perform scaling, offsetting, and limiting.

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 sub-segments properly. There are helper classes that provide constructor methods for the standard envelope types: triangle, AR, ADSR, 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, special "variable" objects are provided by the CSL framework.

There are several other helper classes to support control and parameter interpolation, patch management, and "instrument" graphs with named variables.

Start-up and Configuration

Once started, a CSL-based program can read configuration commands from an input file, from the database, or from CORBA messages; the format is a simple declarative text file, so that various front-end tools can be used to build patches. By default, CSL DSP servers use the "instrument library" they implement to publish an OSC address space, i.e., they take a set of DSP graphs and define the set of OSC network messages they understand to set instrument parameters and trigger envelopes.

Complex configurations of several CSL instances (and additional mixer / spatializers and play programs) can be stored in a database and started up by the Yellow System framework's tools (see below). This framework provides some fault tolerance and runtime load balancing.

CSL Databases

A number of relational and object databases are used by CSL. Instrument spectra are stored in a SHARC (Sandell Harmonic ARChive [Sandell]) timbre database, sound samples can be loaded from a sound database, and the Yellow System framework uses databases to represent the hardware facilities of the available processing network and configuration information about distributed CSL applications. The DPE system also uses a CORBA naming service (a distributed object database) and interface repository at run time.

OSC interface

The default event and parameter control interface in CSL uses the U. C. Berkeley "Open Sound Control" (Wright and Freed) network protocol. There is a customized OSC naming hierarchy for each synthesis technique. An "instrument" object consists of a DSP graph and a list of its "accessors"—the names of its parameters and the functions that one uses to set them. With this, an OSC address space can be generated for any combination of instruments running in a CSL server.

MIDI input is handled by a MIDI-to-OSC translator built by one of the authors (Ramakrishnan).

CORBA Service Interface

The CREATE "Yellow System" is a distributed real-time application management framework (called a distributed processing environment or DPE by some). There are two main components to Yellow: (a) the Real-time Interface Description Language (RIDL [Pope, Engberg, and Holm]), which allows a user to describe the run-time behavior of a server (i.e., a CSL-based program); and (b) the *Service* interface, which defines a set of functions for servers to implement so that they can be managed and monitored by Yellow's tools.

If CSL programs implement these functions, it allows the Yellow manager to administer complex collections of CSL-based servers running on several computers on a network. The goal of this is to support run-time load balancing for large-scale synthesis and spatialization applications, which we call "orchestral-scale sound synthesis" (see Figure 1 above).

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 whereby data packets have a header that incorporates an instance ID and sequence number. CSL servers can then run on 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.

Progressive Code Examples

The following annotated code examples are intended to give the reader a taste of CSL programming, and a quick tour of the CSL class library. As mentioned above, patch editors are planned that will allow non-C++-literate users to configure CSL DSP graphs.

Simple Oscillators and Patching

```
// The simplest example -- play a 220 Hz sine.
// 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);

// Add an amplitude scaler to play the sine wave
//  at 1/4 volume using a variable and a multiplier.
        Sin vox(220);
        StaticVariable amp(0.25);
        MulOp mul(vox, amp);
        io->set_root(mul);

// 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);
```

NB: the *io->set_root()* calls are left out of the remaining examples for brevity.

```
// Perform simple frequency modulation of a 220
// Hz sine wave plus/minus 100 Hz at a 10 Hz
// modulation rate.
        Sin vox, mod;
// Set the modulator’s frequency.
        mod.set_frequency(10);
// Scale the modulator by a constant value.
        DynamicVariable smod(100, mod);
// Create a static variable for the offset.
        StaticVariable offset(220);
// Add the offset to the modulator.
        AddOp fm(offset, smod);
// Set the carrier oscillator’s frequency to be the
// scaled, offset modulator (i.e., do FM).
        vox.set_frequency(fm);

// Make a glissando using a 3-second line
// segment object.
        Sin vox;
// LinSeg constructor: start_val, stop_val, duration
        LineSegment line(110, 8000, 3);
        vox.set_frequency(line);
```

Envelopes

```
// Create and use an ADSR envelope.
        Sin vox(220);
        CO_FLOAT duration = 3.0;      // time
        CO_FLOAT attack = 0.06;       // time
        CO_FLOAT decay = 0.1;         // time
        CO_FLOAT sustain = 0.1;       // value
        CO_FLOAT release = 1.5;       // time
        ADSR adsr(duration, attack, decay,
                    sustain, release);
        MulOp mul(vox, adsr);
// Start a note by triggering the envelope.
        env.trigger();

// Create a breakpoint envelope for use as a
// glissando function.
        Sin vox;
// General-purpose envelope constructor:
//   duration, time, value, time, value, ...
        Envelope env(3, 0, 220, 0.7, 280,
                    1.3, 180, 2.0, 200, 3, 2000);
        vox.set_frequency(env);
        env.trigger();

// FM instrument with different amplitude and
// modulation index envelopes.
// amplitude env = std ADSR.
        ADSR a_env(3, 0.1, 0.1, 0.3, 1);
// index env = fancier.
        Envelope i_env(3, 0, 0, 0.1, 6, 0.2, 1,
                        2.5, 4, 3, 0);
// Declare 2 oscillators.
        Sin vox, mod(110);
// Multiply index envelope by the mod freq.
        DynamicVariable var(110, i_env);
// Scale the modulator by the index envelope.
        MulOp i_mul(mod, var);
// Add in the modulation.
        AddOp adder(220, i_mul);
// Set the carrier's frequency.
        vox.set_frequency(adder);
// Scale the carrier by the amplitude envelope.
        MulOp a_mul(vox, a_env);
// Play a note by resetting the envelopes
        a_env.trigger();
        i_env.trigger();
```

Processing and Filtering

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

// Apply a band-pass filter (300 - 700 Hz
//   [= 500 +- 200]) to pink noise.
        PinkNoise pnoise (20000);
        ButterworthFilter filter(pnoise,
                    pnoise.rate(),
                    Filter::BAND_PASS,
                    500, 200);

// Really slow additive synthesis -- add 4 scaled
// sine oscillators in a mixer.
        Sin vox1(431);      // create 4 scaled sines.
        MulOp mul1(vox1, 0.3);
        Sin vox2(540);
        MulOp mul2(vox2, 0.1);
        Sin vox3(890);
        MulOp mul3(vox3, 0.3);
        Sin vox4(1280);
        MulOp mul4(vox4, 0.01);
        Mixer mix(2);       // create a stereo mixer
// Mix the oscillator and 3 scaling multipliers.
        mix.add_input(mul1);
        mix.add_input(mul2);
        mix.add_input(mul3);
        mix.add_input(mul4);
```

Reading and Playing a Sound File

```
  // Load a sound file into memory.
        SoundFile fi("kombination1a.snd");
```

Spectral Processing

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

  // Create two spectra and cross-fade
  // i.e., vector IFFT synthesis.
        IFFT vox1, vox2;
        Spectrum spectrum1, spectrum2;
  // Set some data in the first spectrum
        spectrum1.set_partial(3, 0.25, 0);
        spectrum1.set_partial(5, 0.25, 0);
        vox1.set_spectrum(spectrum1);
  // Add some partials to the 2nd spectrum
        spectrum2.set_partial(30, 0.25, 0);
        spectrum2.set_partial(40, 0.25, 0);
        vox2.set_spectrum(spectrum2);
  // Create inverse line envelopes
        LineSegment env1(1, 0, 3);
        LineSegment env2(0, 1, 3);
  // Cross-fade between the two spectra
        MulOp mul1(vox1, env1);
        MulOp mul2(vox2, env2);
  // Add the two components together
        AddOp add3(mul1, mul2);
```

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 floats 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.

The CO.h Header and Utility Classes

The header file CSL_Types.h is included in all CSL source files and contains useful data type macros (e.g., sample, buffer) so that the source code can be more flexible and platform independent.

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

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.

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 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.

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 simper. One could certainly write a multichannel unit generator as a subclass of FrameStream.

CSL Mix-in Classes

There are several other base classes that are used as “mix-ins” in multiple inheritance with FrameStreams.

Processor classes generally take one or more inputs that are FrameStreams (or entire DSP subgraphs). They forward the *next_buffer()* calls they receive from the lower levels of the DSP tree to their inputs, and then do some manipulation on the sample buffers.

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

The Envelope Class Hierarchy

Control functions are often most-simply described as break-point functions. Interpolation between breakpoints can be linear, exponential, cubic, or use other interpolation algorithms. In CSL, there are special envelope classes for traditional kinds of envelopes, such as attack / decay / sustain / release (ADSR) generators.

The Variable Class Hierarchy

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

The IO Class Hierarchy

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

RemoteFrameStreams and UDP_IO

A RemoteFrameStream is a FrameStream that is connected by a UDP network 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 listens 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.

The Instrument Class Hierarchy

There are several utility classes to make it easier to manage DSP graphs. A Instrument object has a DSP graph, a set of 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);
list[2] = new Accessor("in", set_index_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
    /i1/in:     set-index command
```

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, or from score file readers.

CSL *main()* Functions

There are several ways to compile CSL programs, and several versions of the *main()* function to be used for CSL programs. The simplest one of these 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.

Since CSL is 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.

Applications

Starting in the Winter of 2002, we have used CSL for several applications.

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) 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 April or 2003, ran very reliably, 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.

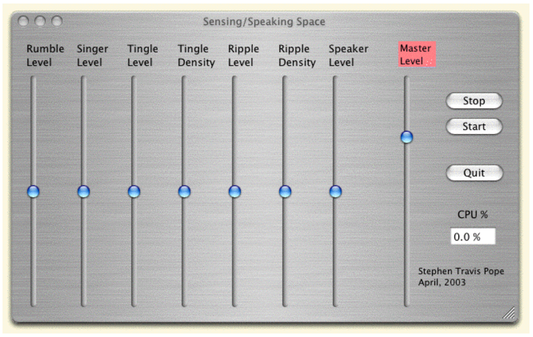

Figure 3: Sensing / Speaking Space GUI

Onde Corner

...text / figure from CR...

Reverb Plug-in

In a graduate course on spatial sound, students developed a series of panners and reverberators based on various simplified versions of the CSL framework. Later, CSL was used to make a plug-in for the Steinberg VST API that implements a convolution-based reverberator.

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 Yellow System infrastructure.

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).

Acknowledgments

The team that participated in the MAT 240D course in the Spring of 2002 all contributed to the current CSL system. The design and basic C++ framework were provided by the first author of this paper, and major additions and refinements were implemented by Chandrasekhar Ramakrishnan, Larry Miller, John Goforth, Derek Piasecki, Brent Lehman, and Doug McCoy.

References

Burke. JSyn

Burke, Bencina, et al.: PortAudio

Cook: STK

Lansky. CMix

McCartney. SuperCollider

OMG: CORBA

Pope. Three Languages

Pope. Siren

Pope et al.: DPE

Pope, Engberg, and Holm: DPE RIDL

Puckette: Max

Sandell: SHARC

Scaletti: Kyma

Schottstaedt: CommonLispMusic

Wright. OpenSoundControl

Contact: Stephen T. Pope
stp@create.ucsb.edu
Home Page
http://create.ucsb.edu/CSL
FTP Site
ftp://create.ucsb.edu/pub/CSL

Appendix: C++ Code Examples

In the examples, below, we present the source code for a simple CSL "unit generator" class: a sawtooth oscillator.