CSL (sizzle): The CREATE Signal Library

Stephen Travis Pope
Center for Research in Electronic Art Technology (CREATE)
Graduate Program in Media Arts and Technologies (MAT)
University of California, Santa Barbara (UCSB)
stp@create.mat.ucsb.edu
June, 2004

Outline: Intro. to CSL

• CSL ("Sizzle"): The CREATE Signal Library for digital audio synthesis & processing
  - Context: CREATE R&D
  - Background and relatives
  - Technical overview
  - Code examples
  - Evaluation, next steps
  - Steps for getting started programming with CSL

CREATE Synthesis/Performance Group Goals

• Support reliable “orchestra-scale” sound synthesis, multi-modal gestural sensing and control, and pluriphonic projection (up to 128 channel output in the CNSI sphere)

What’s CSL?

• General-purpose, portable C++ programming framework for real-time digital audio synthesis and processing
• Used for stand-alone applications, plug-ins, OSC/MIDI servers, etc.

CSL Relatives (Software Synthesis)

• Like Cmix, STK, Siren, JSyn, MxV, or CLM
  – Delivered as a fcn/class library in a general-purpose programming language
• Unlike SuperCollider, Csound, Max
  – Not its own language
  – No scheduler
  – Uses C++ development environment
Why on Earth another one???

- Cmix -- old, flaky
- SuperCollider -- different question, complex
- Csound, Music-N -- not languages, source clarity
- Jsyn -- closed DSP kernel
- STK -- PM-centric, tick model
- CLAM -- way complex
- CLM -- who knows LISP?
- Siren/Squeak -- who knows Smalltalk?

Our Requirements
- Simple, easy to learn
- Flexible, multi-purpose
- Portable
- Scalable
- Embeddable
- Distributable
- Network-oriented
- Debuggable

CSL Background
- "CREATE Oscillator" -- 1998, CORBA_A/V-based sample streaming, CORBA IDL for instruments
- MAT 240D course (digital audio synth. techniques, Spring '01, '03)
  - CO1 (minimal 1 KLOC), CO2 (full-featured)
  - CSL_lean (redesign from scratch by one person)
  - CSL3 (2004, 25 KLOC, full-featured)
- Designs driven by immediate needs for concrete applications (pieces, theses, etc.)

CSL3 Basics: Core Classes
- **Buffer** objects (1 class + helpers)
  - Multichannel (non-class-leaved) sample storage
  - "Smart" object (not just a float*), ptr. mgmnt.
  - Handle malloc/free, filling statistics, etc.
- **FrameStream** classes (Ugens) (many)
  - Respond to the message next_buffer(input, output)
  - Processors have a FrameStream as input
- **Mix-in** classes (vs. wrapper classes)
  - Phased, Positionable, Writeable, Cacheable, etc.

Simplest CSL Program

Sine wave with envelope

// Create a sine oscillator -- this is a C++ comment
Sine oscc(220.0);  // freq = 220 Hz

// Create an ADSR envelope -- args are (dur, att, dec, sus, rel)
ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);

// Create a multiplier for osc & env
MulOp mul(osc, env);

// Plug it into the (global) output
globalI0.set_root(mul);

Reverb’d Panning Dual-Env. FM

(C’Ugens, minimal, procedural style)

// FM instrument with stereo panning and reverb
ADSR a_env(1, 0.01, 0.1, 0.1, 0.6);  // create amp env, ADSR(time, a, d, s, r)
ADSR l_env(1, 0.001, 0.1, 0.5, 0.5);  // create FM mod. index env.
l_env.set_scale(110);  // scale l_env by base freq.
Sine sin = mod(110);  // create 2 sine oscs: carrier & modulator
mod.set_scale(a_env);  // scale the modulator by the a_env
mod.set_offset(l_env);  // add in the base freq.

car.set_frequency(mod);  // set the carrier’s freq, to the modulator
a_env.set_input(car);  // plug the carrier into the a_env’s input
Sine pos(0.25);  // create an LFO for panning
Panner paner = env;  // plug the a_env into a stereo panner
StereoVerb svd;  // plug the panner into a stereo reverb
GlobalOut root(verb);  // plug the reverb into the output
// glitz-off, input(verb);  // or add it to a global mixer

Sine Osc Alternatives

// Use the envelope object as a generator and processor (VCA)
SumOfSines osc(220.0, 1, 5, 0.7...);  // make a sum-of-sines
Triangle env(3.0);  // triangle envelope
env.set_input(osc);  // send osc as input to env
env.set_root(env);  // env is root

// Use the osc’s scale (volume control or AM) input
SquareBk osc(220.0);  // make a band-lin square
Gaussian env(3.0, 0.2);  // envelope with bell width
osc.set_scale(env);  // set osc scale to env
osc.set_root(env);  // osc is root
The Big Picture of CSL

- Your basic DSP graph
- Now connect it to control input (OSC, MIDI, GUI, CORBA, XML), and audio IO driver object
- Buffering and latency tuning

CSL FrameStream Details

- Core FrameStream methods
  - `next_buffer(inbuf, outbuf)` - fill in a buffer's worth of frames (input buffer is signal from ADC)
  - `next_sample(inbuf, outBuf)` - 1 sample; adjust phases
  - `is_fixed_over(in)` - is the receiver's value fixed over range?
  - `is_active()` - are a graph's envelopes on?
- Several policies for handling `next_buffer()` with multi-channel I/O buffers: call `mono_next_buffer()` and iterate (vs. copy - FanOut and Splitter/Joiner)

Swept Band-pass Filtered Noise

```c
void test_filter_sweep() {
    ADSR a_env(3, 0.1, 0.3, 1); // ampl env = std ADSR
    WhiteNoise wnoise; // noise generator
    Sine centerSweep(0.5, 500.0, 1000.0); // args = frq, ampl, offset
    Sine BWsweep(0, 50.0, 200.0); // args = ctr, bw
    Butter ffilter(wnoise, // BPF: in, type, ctr, bw
                   Butter::BAND_PASS, centerSweep, BWsweep);
    a_env.set_input(ffilter);
    log_msg("playing filter_sweep...");
    gMixer->add_input(a_env); // add to global mixer
}
```

SoundFile Playback Loop in a Thread

```c
// Function that plays random samples from the given sound library
// Sample Libraries look like `vector <SoundCue * > snap;`
void *play_sound(void *args) { // Signature for linking as a thread
    SoundCue *voice; // Sound cue pointer
    StaticVariable pos(); // Panner position value
    Panner pan; // voice, pos);
    gMixer->add_input(pan);
    while (true) { // Loop playing sounds
        voice = snap[rand() % snap.size()]; // Get a sound cue from the library
        pan.pos.value(voice->pos()); // Send it to panner
        pan.pos.value(voice->pos()); // Set a new position
        voice->trigger(); // Now trigger the sample
        thread_sleep(voice->duration() / 44100) + (f_rand() * 12); // Sleep a bit
    }
}
```

CSL DSP Graph Flexibility

- Sub-graphs can run at different:
  - Sample rates (for control),
  - Buffer sizes (for transforms),
  - Numbers of channels (for efficiency),
  - Buffer formats (interleaved or not),
  - In different threads, etc.
- These can be changed (within reason) at runtime (e.g., for load- or traffic-balancing)
Multi-host CSL Graphs

- Distributed sub-graph processing with RemoteIO (server) and RemoteFrameStream (RFS, client)
- RFS protocol, (optional) client buffering

RemoteIO, root of graph, server
CSL RFS protocol (over TCP, UDP, ATM)
RemoteFrameStream, node, client

RemoteStream/RemoteIO Details

- Uses simple protocol, LAN-oriented (we use switched 1000BaseT & TCP)
- Relatively careful (packet header/trailer, sequence numbers, format packets)
- Double-send optional with UDP/ATM
- RFS client uses ThreadedFrameStream with variable-sized (zero-possible) RingBuffer

Using RemoteFrameStreams

- Server (sample source) side: IO is RemoteIO
gIO = new RemoteIO(the_port); // Socket-based IO object
gIO->open(); // open client socket
gIO->start(); // start server read thread
// server CSL patch follows

- Client (sample reader) side
RemoteStream rfs("host_name", the_port, 2, buf_size);
Stereoverb verb(rfs); // reverberate the RFS (e.g.)
gIO->set_root(verb); // plug reverb into the (real) output

Control, Latency, Scheduling

- All CSL processing is triggered by output requests (pull model, buffer size determines control rate)
- Slow computations should use ThreadedFrameStreams or transform/convolver threads
- Control may change asynchronously; query is_processing() (optional semantics of control)
- Latency determined by buffer size, amount of caching in graphs, and RFS remote links (few msec for small buffers, < 1 msec doable ?)
- Dynamic graphs are rare; no time or event models

Instruments and OSC/MIDI/XML

- Instrument object
  - Holds onto a DSP graph; adds “reflective” accessors
  - Server main() function loads an instrument library, generates OSC address space or MIDI map (from accessors), and starts a listener thread on a socket
  - Example:
  
  ```
  // C++ instrument accessor decl.
  int[i] = new Accessor("du", set_duration_f, CSL_FLOAT_TYPE);
  int[i] = new Accessor("am", set_amplitude_f, CSL_FLOAT_TYPE);
  ... results in OSC address space
  int[i] // instrument i OSC commands
  int[i]: // set-duration command
  int[i]: // set-amplitude command
  ```

GestureSensor Drivers & Servers

- Reusable sensor driver framework
  - Serial in, caching/differencing/throttling, OSC out
- GestureSensors: receive OSC or MIDI
  - void * mData; // data array (typically a float *)
  - char * mCmd; // OSC command (without the '/')
  - char * mTypeString; // OSC type string, e.g., "ffff"
  - Event input thread mgmnt
  - Parsing and differencing
  - Map to static or global data or messages
- Subclasses
  - Gloves, Ebeam, Matrix, FOBirds, AdC_Panner, etc.
CSL main() for OSC Processing

```cpp
// Set up OSC address space
init_OSC_addr_space();
// EITHER: add the instrument library OSC addr space
setup_OSC_instr_library(instrLibrary, numInstrument);
// OR: create a background thread for a GestureSensor
Thread * aThread = Thread(Thread::MakeThread(fcn);
aThread->fork_thread(sensorThreadFcn, &someArgument);
// Start the I/O callback thread for the global IO
gIO->start();
// Run the OSC I/O loop function (never returns)
mainOSC_loop(thisUDPPort);
```

CSL Cross-platform Portability

- MacOSX/Xcode, *nix, Linux/KDevelop, MS-Windows/VisualStudio
- Cross-platform APIs
  - PortAudio for RT sound IO
  - LibRtsFile for real-time sound file IO
  - PortMDI for MIDs
  - LibNewRts for probability distributions
  - FFTW for FFT
  - CyberXID for VRML, OpenGL
- Issues
  - C++ compiler, socket/thread code, GUI
  - Base sample data type (float vs int)
  † = may use platform-specific APIs (ConvAudio, DSP_FFT, etc.)

CSL "beep" main (all of it!)

```cpp
// Beep_main.cpp -- the simplest CSL 'main' program -- a 3-second beep
#include "CSL_Audio.h"
#include "CSL_Synth.h"
using namespace os;
// CSL 'kitchen sink' includes
int main(int argc, const char * argv[]) { 
    gIO gIO();
    // PortAudio I/O object
    if (! gIO.gIO("beep.aiff", Size::error))
        // OR: use a File I/O object
        create a sine oscillator at 600 Hz
    gIO.open_file();
    // plug it in to the IO
    gIO.Open();
    sleep(1000);
    // sleep 3 seconds (CSL std in ms)
    gIO.sleep();
    gIO.close();
    // close the IO
    exit;
} // link this with CSL.lib, PortAudio, etc.
```

OSC with a Shell Script

```bash
# Shell script to test sending OSC messages to a simple CSL server
# Create a convenient shell command alias
alias osc="sendOSC & localhost 54321"
# Play a note ("p" command) on instrument 1 (fn) and sleep
 osc fn 1; sleep 3
# Set a new "cf" value and play a note on instr 2
 osc cf; osc fn 2; sleep 3
# play an FM note with parameters dac/amp/cn/mod/ind
 osc fn 4j,4,0,0,3,220,0,357,4,3; sleep 4
# load a sound file in instr 6
 osc fi,4,0,0,3,220,0,357,4,3; sleep 1
# play a sampled sound with it
 osc fi,4,0,0,3,220,0,357,4,3; sleep 1
```

Using CSL

- As a library
  - Link a graph and IO into your application, game, GUI, etc.
- For plug-ins
  - AudioUnits or VST with GUIs (call back to next_buffer())
- For OSC, MIDI, CORBA, XML-RPC, etc. servers
  - Stand-alone instrument groups as soft-synths, RemoteIO
- With CRAM
  - Multi-host control / server / output configurations
- The main function creates graph or mixer, may spawn threads, then registers an IO call-back object

CSL Example: Se/Sp_Sp (2002)

Camera-based multi-user sensing (aware space)

Computer vision SW follows movements and filtering among attended; sends OSC mgs to 3 (Video, CSL) servers

Synchronized multi-camera projection and 6-ch. surround sound

Port from SC2 to CSL2

Sensing, computation, multi-presentation (MVC)
Example: OnDeCorner

- CR’s AudioUnit plug-in for experimenting with wavelet transforms
- Pluggable FWT code
- Play to DAC or file

Example: Ouroboros

- CR’s AudioUnit host application for processing sound files and live input
- Extensions planned for remote AudioUnits

Example: Expert Mastering Assistant

Process: Analysis, GenreDB, Mapper, DSP, Interact

Generating CSL Graphs/Events

- Using scripting languages
- Smalltalk Slang translator
- From XML
- DragNDrop “patcher” GUIs
- Storing signals and graphs in an OODB
- Instrument libraries and event stores
- Auto-gen of flat namespace for C RMI

Example: LUA Patcher (worked, but failed)

CSL as a library for a scripting language

```lua
-- Lua program for a panning chaotic oscillator
panning_chaos = function ()
  lorenz = Lorenz();
  envargs = {0.5, 0.0, 0.0, 0.003, 0.5, 0.5, 0.0};
  envelope = Envelope(envargs);
  panner = Panner2(lorenz, envelope);
  audio_out(panner);
end
```

So we know it all, right?

- NOT!
  - Many open architecture, design, modeling, implementation, deployment, issues
  - Some basic choices we’re still debating
  - Some real dilemmas, limitations, principles
  - Tensions between our design bias towards simplicity and “creeping featurism”
Open CSL Design Issues

- Basic models: buffer-based, event-based, signal-based
  - Current pull-model driven by PortAudio and CoreAudio APIs; granularity of events
  - Need a unification of types (semantics) of buffers (samples, FFT frames, FWT frames, IRs, etc.)
  - Signal semantics: operators on buffers vs. procedural ugens?
- How to support dynamic graphs in a simple system (punt)
- That latency thing, polynomial ctrl interpolation, clock sync.

Conclusions

- For our requirements, we really had to start from scratch for most of the components.
- The KISS principle (or XP) paid off in simplicity, flexibility, and ease of use.
- There are many things we could have done other ways (we’re still debating; that’s the whole fun of it!).
- See create.ucsb.edu/ [Siren, CSL, CRAM]

CSL Source Organization (Categories)

- Main - Test/demo main() driver functions
- Kernel - Buffers and FrameStreams
- Sources - Oscillators, noise, envelopes, PhysMod
- Processors - Operators, filters, mixers, panners
- IO - IO drivers and LAN streaming
- Utilities - Thread and buffer support classes
- Instruments - OSC/MIDI instrument wrappers
- QT_GUI - Signal view GUI support for QT widgets
- OSC - CNMAT OSC library
- Auralizer - N-channel convolution-based spatializer
- Documentation - README, etc.

Speed Hacks & Optimizations

- User-visible optimizations
  - is_fixed_over(), is_active() – used
  - is_linear_over(), is_polynomial_over() – ?
- Several kinds of buffers (cache optim.)
- Control interpolation?
- DSP graph-to-SMP allocation
- Managed sample-rate conversion
- Better C++ compiler (IBM or Intel / AMD)
- Many interesting optimizations would greatly complicate the system (our guess)

Getting Started Using CSL

- Download zip file or tarball (or subversion/cvs tree)
- Read the README and on-line docs
- Install support libraries (PortAudio, PortMIDI, libSndFile, OSC, FFTW, libnewran, etc.)
- Open C++ project tool for your platform (Xcode, KDevelop, VisualAudio, V) (Makefile)
- Select target main() file
  - Basic demo (start here to make certain you can link & run)
  - Test_mains (edit end of file and run)
  - OSC server, MIDI softsynth, other main()s
- Build CSL kernel libraries and demo target
- Start debugger and run!

Central CSL Header Files

- CSL_Type.h – the main include file for CSL3: data typedefs and cross-platform macros
- CSL_Cnch – the CSL Kernel: Buffer, FrameStream, SampleStream, UnitGenerator, MixIn classes
- Gestalt.h – class CGestalt (system constants)
- Variable.h – abstract external variable (plug) class
- Oscillator.h – specification of the base oscillator class and standard waveform generators
- Envelope.h – The breakpoint envelope classes
Writing a CSL FrameStream Class

```cpp
// Sawtooth oscillator class specification (h file)
class Sawtooth : public Oscillator {
  // declare class protected:
  status mono_nextBuffer(Buffer & inB, Buffer & outB, unsigned inBNum, unsigned outBNum);
  public:
    // constructors
    Sawtooth();
    Sawtooth(float frequency);
  }

  // Writing the next_buffer() method
  Phonclass Oscillator - Sawtooth Phar

  // Class Hierarchy
  FrameStream - SampleStream - UnitGenerator

  // Status
  status status;

  // Constructor
  Sawtooth(float frequency);

  // Method
  status mono_nextBuffer(Buffer & inB, Buffer & outB, unsigned inBNum, unsigned outBNum);

  // Sawtooth mono_next_buffer() Example
  status Sawtooth::mono_next_buffer(Buffer & inB, Buffer & outB, unsigned inBNum, unsigned outBNum)
  {
    if (sample * bufptr = outB.numBuffers[outNum];)
      // all ptr of out
      unsigned numFr = outputB.numFrames;
      // # of frames requested
      float rateRecip = 1.0 / sampleRate;
      // main sample loop
      if (unsigned i = 0, i < numFr, i++)
        "buff ptr = (phas eAcc "scaleC) + offsetC;" store value to buffer
        if(_phas eAcc>=1.0)
          // reset phase
          _phaseAcc = 0.0;
      return cslOk; // return OK status
  }
```

CSL Add-on Packages

- "Advanced" sources
  - SHARC/IFFT additive synthesis, physical models/FDN,
  - granulators, waveshapers
  - GestureSensor drivers and OSC mapping
  - OSC and CSL instruments
  - Auralizer
    - VRML-based geometry, late-reverb modeling, and low-latency distributed
      many-channel convolution
  - HRTF FIRs and HRTF databases (used with OSC head trackers)
  - QT GUIs signal display, control monitoring
  - CRM Interface: CRAM manager service class for CSL servers
  - Wavelet code: wave++ discrete wavelet transform

CSL Home Page
http://create.ucsb.edu/CSL

CSL Resources

- CSL Downloads (doc, source tarball)
  http://wwwcreate.ucsb.edu/CSL/CSL_Overview.pdf
http://wwwcreate.ucsb.edu/CSL/CSL_tgz

- CSL Mailing List
  http://create.ucsb.edu/mailman/listinfo/CSL
  Send to CSL<create.ucsb.edu

- Related Projects at CREATE
  - Auralizer & VRML
  - Pulsar Generator
  - Creatovox
  - MusicVisualization
  - FMAK DB
  - TimeMachine
  - InteractEMGroup
  - Creatophone
  - Time-D Decomp
  - SC 3 Work

- CSL Processes

  - MixIn class Processor adds an input SampleStream
  - next_buffer method calls Processor::pull_input (inB, outB), possibly using a temp buffer
  - This calls input’s next_buffer method
  - Now the processor operates on source’s input buffer into its output buffer
  - Filters, panners, etc.

- CSL Processors

  - Sawtooth
    - Sawtooth mono_next_buffer() Example:
      ```cpp
      status Sawtooth::mono_next_buffer(Buffer & inB, Buffer & outB, unsigned inBNum, unsigned outBNum) {
        if (sample * bufptr = outB_numBuffers[outNum];)
          // all ptr of out
          unsigned numFr = outputB_numFrames;
          // main sample loop
          if (unsigned i = 0, i < numFr, i++)
            "buff ptr = (phas eAcc "scaleC) + offsetC;" store value to buffer
            if (_phas eAcc >= 1.0)
              // reset phase
              _phas eAcc = 0.0;
        return cslOk; // return OK status
      }
      ```