SW R&D @ CREATE: Siren, CSL, and CRAM

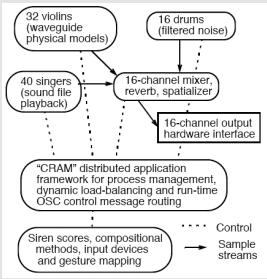
Stephen Travis Pope
Center for Research in Electronic Art Technology (CREATE)
Graduate Program in Media Arts and Technologies (MAT)
University of California, Santa Barbara (UCSB)
stp@create.ucsb.edu

Overview

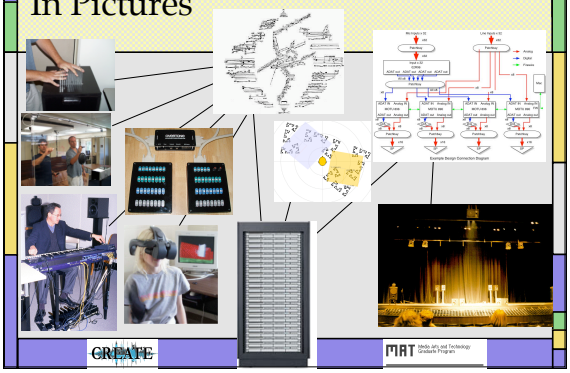
- The Context: Siren, CSL, and CRAM
- Siren: Object models, control mappings, and GUIs for composers and performers
- CSL ("Sizzle"): The CREATE Signal Library for distributed DASP
- CRAM: Managing distributed real-time applications
- Examples, evaluation, next steps

Goals

- Support "orchestra-scale" sound synthesis, multi-modal gestural sensing and control, and pluriphonic projection



In Pictures



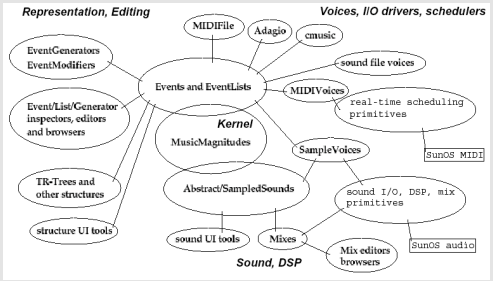
What's Needed?

- Representations and interfaces for composers and performers
- Scalable DASP framework
- Management infrastructure for distributed RT software

Siren

- Smalltalk-based framework for sound/music processing under development since 1984
- Focus on structure, control, and composition, rather than on DSP or notation
- Platform for music representation language development

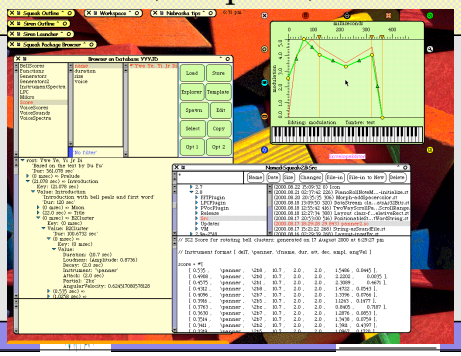
Siren Components



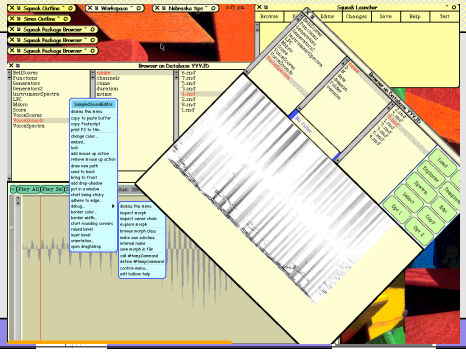
What's Siren?

- "Smoke" music representation language (music magnitudes, events, event lists, generators, modifiers, …)
- I/O "voices" (players, drivers, property/parameter mappers) for many formats
- Control I/O and routing for OSC and MIDI
- GUI widgets and apps for music
- OODBMS interfaces

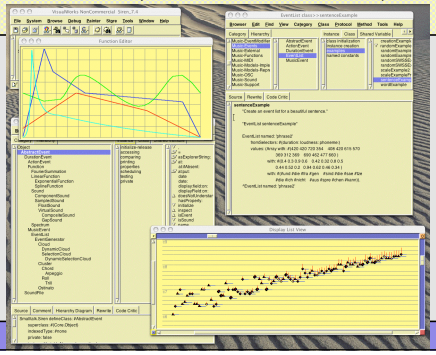
Siren 2002 (in Squeak)



Siren GUI Tools



Siren 2003 (in VisualWorks)

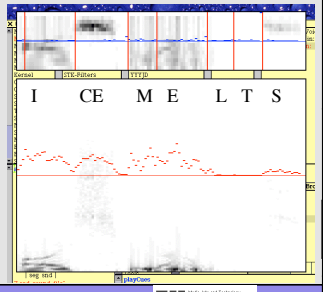


Composition in Siren

```
RotatingBellCluster class methodsFor: 'HeavenEverywhere' stamp: 'stp 7/28/2003 18:11'
setup: dataSelector functions: fcnSelector
    "(RotatingBellCluster setup: #b2Data functions: #b2aFunctions) eventList explore"

    | i dat parts clust fcns |
    parts := OrderedCollection new: 16.                "Load partial data"
    dat := self perform: dataSelector.
    i := 1.
    [i < dat size] whileTrue:
        [parts addFirst: (BellPartial named: (dat at: i) freq: (dat at: i + 1) ampl: (dat at: i + 2) dur: 10.7).
        i := i + 3].
    clust := self new partials: parts.             "Load function data"
    fcns := self perform: fcnSelector.
    clust spectralFcn: (fcns at: 1).               "Spectral weighting function"
    clust densityFcn: (fcns at: 2).                "Time/density function"
    clust velocityFcn: (fcns at: 3).               "Mapping of pitch to angular velocity"
    clust att: 2.0 sec.                            "Set bell cluster parameters"
    clust dec: 2.0 sec.
    clust duration: 90.0.
    clust pitch: 14.07 Hz.                         "= Base * 1.61803398"
    clust ampl: -7.2 dB.                           "mezzopiano"
    clust voice: #panner2.
    clust name: #'rBell.b2.14c'.
    ^clust                                         "Answer bell cluster"
```

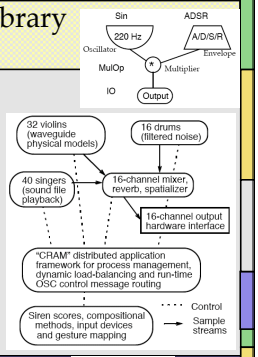
Siren Applications

- 8S speech segmenter and database
- SeSpSp and EMA prototypes
- Recent pieces (see below)



The CREATE Signal Library (CSL, "sizzle")

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



CSL Relatives

- Like Cmix, STK, JSyn, or CLM
  - Delivered as a library in a general-purpose programming language
- Unlike SuperCollider, Csound
  - Not its own language
  - No scheduler
  - Uses C++ development environment

Why on Earth???

- Cmix
  - Old, flaky
- SuperCollider
  - Answers a different question
- Csound, Music-N
  - Not languages, source clarity
- JSyn
  - Closed DSP kernel
- STK
  - PM-centric, not mutli-channel
- CLM
  - Who knows LISP?
- Siren/Squeak
  - Who knows Smalltalk?

Our Requirements

- Easy to learn
- Flexible
- Portable
- Scalable
- Distributed
- Network-oriented
- Debuggable

CSL Background

- "CREATE Oscillator" dating back to 1998
- MAT 240D course (digital audio synthesis techniques, Spring '02)
  - CO1, 2
  - CSL_lean (redesign from scratch by one person)
  - CSL3
- Designs driven by immediate needs for concrete applications

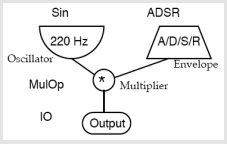
CSL Basics

- Buffer objects
  - Multichannel non-interleaved sample storage
  - "Smart" object, not just a (float **)
- FrameStream classes (Ugens)
  - Respond to the message `next_buffer(input, output)`
- Processors
  - Have a FrameStream as input
- Mix-in classes
  - Phased, Positionable, Writeable, Cacheable, etc.

Canonical CSL Example

Sine wave with envelope

```
// Create a sine oscillator
        Sine vox(220.0);
// Create an ADSR envelope
        ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
// Create a multiplier
        MulOp mul(vox, env);
// Plug it into the output driver
        gIO.set_root(mul);
```



Sine Osc Alternatives

```
// Simplified sine+envelope example using the envelope
// object as a combined generator and amplifier
   Sin vox(220.0);
   ADSR env(3.0, 0.06, 0.2, 0.2, 1.5);
   env.set_input(vox);
   gIO.set_root(env);


// Same 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);
   gIO.set_root(vox);
```

Less Trivial CSL Example

```
ADSR a_env(1, 0.01, 0.1, 0.1, 0.6);     // amplitude env = std ADSR
a_env.scale_values(0.25);               // scale it to quiet
ADSR i_env(1, 0.001, 0.1, 0.5, 0.5);    // index env
Sine vox, mod(110);                     // declare 2 oscillators
DynamicVariable var(110, i_env);        // multiply index env by mod freq
MulOp i_mul(mod, var);                  // scale the modulator by the env
AddOp adder(i_mul, 220.0);              // add in the modulation
vox.set_frequency(adder);               // set the carrier's frequency
MulOp a_mul(vox, a_env);                // scale the carrier's output by the
amplitude envelope
Sine pos(0.25);                         // LFO panning
Panner pan(a_mul, pos);
Stereoverb verb(pan);                   // add some stereo reverb
gIO->set_root(verb);                    // plug the graph into the output
```

FrameStream Details

- FrameStream
  - next_buffer() - fill in a buffer's worth of frames
  - next_sample() - answer just one sample value
  - is_fixed_over() - answer whether the receiver's value is fixed for a given range of samples
- Policies for handling next_buffer() with multi-channel I/O buffers
  - Call mono_next_buffer() and iterate/copy

Sources, Controls, and Processors

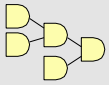
- Sources
  - Oscillators, Noise, SoundFile, Chaotic, IFFT, Physical Models, Granulators, Windows
- Control
  - Envelopes, DynamicVariables, LFNoise
- Processors
  - Operators, Mixers, Filters, Reverb, Panners, etc.
- Support Classes
  - RingBuffer, BlockResizer, ThreadedFrameStream, Interleaver/Deinterleaver, Splitter/Joiners, FanOut

CSL I/O

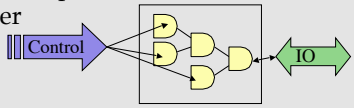
- IO Object
  - Holds onto a DSP graph (the root node, may be a Mixer)
  - Receives periodic call-backs from somewhere (driver such as PortAudio, CoreAudio, UDP, file I/O scheduler, etc.)
  - Sends next_buffer() to the root of its graph and does something with the resulting buffers

The Big Picture

- Basic DSP graph

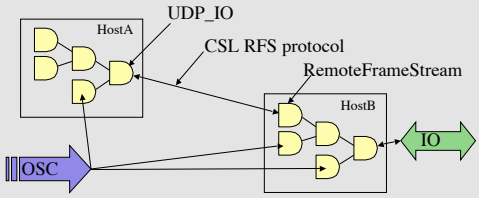


- Connected to control (OSC, MIDI, or GUI) input and HW output driver



Multi-host Graphs

- Distributed processing in CSL



RemoteFrameStream Details

- Uses simple UDP-based protocol
- Relatively careful (packet header/trailer, sequence numbers)
- Double-send optional
- RFS uses ThreadedFrameStream with variable-sized RingBuffer
- No tunneling to date

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), etc.
- These can be changed (within reason) at run-time

Control, Latency, Scheduling

- All CSL processing is triggered by output requests (pull model)
- Slow computations should use ThreadedFrameStreams
- Control can change asynchronously
- Latency is determined by buffer size and amount of cacheing within a graph (small numbers of msec for small buffer sizes)
- Dynamic graphs are rare

Instruments and OSC

- Instrument object
  - Holds onto a DSP graph; adds "reflective" accessors
  - These are used to generate OSC address spaces or MIDI maps
  - Server main() function loads an instrument library and publishes an address space
  - Example

```
list[0] = new Accessor("du", set_duration_f, CSL_FLOAT_TYPE);
list[1] = new Accessor("am", set_amplitude_f, CSL_FLOAT_TYPE);
-->   /i1/                  instrument 1
                    /i1/du:   set-duration command
                    /i1/am:   set-amplitude command
```

CSP Portability

- Cross-platform APIs
  - PortAudio for RT sound IO
  - LibSndFile for sound file IO
  - PortMIDI for MIDI
  - LibNewRan for probability distributions
  - FFTW for FFT
- "Issues"
  - Socket/thread code
  - Base sample data type (float vs int)

Using CSL

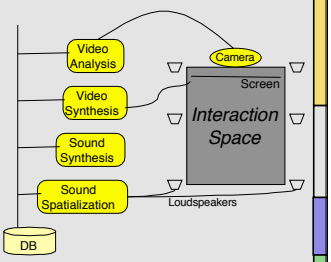
- As a library
  - Link it into your application and GUI
- For plug-ins
  - AudioUnits or VST
- For OSC servers
  - Stand-alone instrument groups
- With CRAM
  - See below
- The main() function just has to register a call-back for the I/O driver

CSL "beep" main

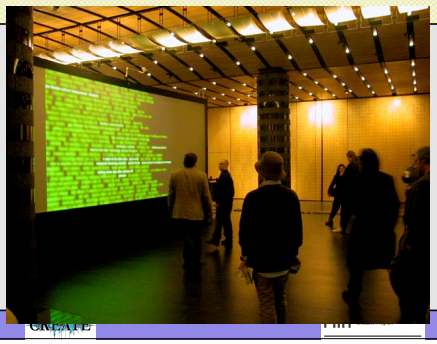
```
//    Beep_main.cpp -- the simplest CSL "main" program -- a 3-second beep
#include "CSL_All.h"                    // CSL "kitchen sink" include
using namespace csl;
PAIO gIO;                               // PortAudio IO object
//FileIO gIO("csl_test.aiff");          // File IO object

                                        // MAIN -- plays a 3-second beep
int main (int argc, const char * argv[]) {
    Sine vox(220);                      // create a sine oscillator at 220 Hz
    gIO.set_root(vox);                  // plug it in to the IO
    gIO.open(); gIO.start();            // open/start the IO
    sleep_sec(3);                       // sleep 3 seconds
    gIO.stop(); gIO.close();            // stop/close the IO
    exit(0);
}
```

Example: Se/Sp_Sp (2002)

- Sensing, computation, (multiple) presentation (MVC or DSCP)
- Camera-based multi-user sensing (aware space)
- Computer vision SW for object model of grouping among attendees
- Distributed real-time network protocols for synchronized multi-camera projection and 6-channel surround sound

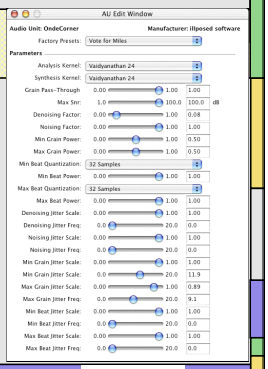


Sensing/Speaking Space @ SFMOMA



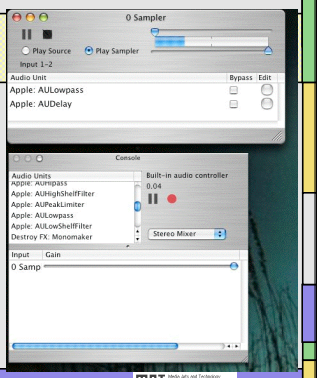
Example: OnDeCorner

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



Example: Ouroboros

- CR'sAudioUnit host application for processing sound files and live input
- Extensions planned for remote AudioUnits



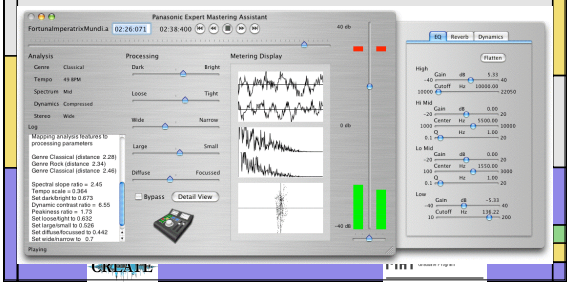
Example: LUA Patcher

CSL as a library for a scripting language

```
-- 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
```

Example: Expert Mastering Assistant

Process: Analyze, Map, DSP, Interact



So we know it all, right?

- NOT!
- Many open design and implementation issues
- Some basic design choices we're still unsure about
- Some real dilemmas
- Tensions between the design bias towards simplicity and "creeping featurism"

Design Issues

- Basic models: buffer-based, event-based, signal-based
  - Current model driven by PortAudio and CoreAudi APIs
  - Need a unification of different types (semantics) of buffers (samples, FFT frames, FWT frames, etc,)
- Whether/how to support dynamic graphs in a simple system

Speed Hacks

- User-visible optimizations
  - is_fixed_over() -- used
  - is_linear_over() -- ?
  - is_polynomial_over() -- ?
- Several kinds of buffers
- Control interpolation? (vs potential clicks on update during sample loop)

Potential Optimizations

- Managed buffer allocation
- Managed sample-rate conversion
- Planning for cache optimization
- Better C++ compiler (IBM or Intel)
- Most of these would greatly complicate the system (our guess)

Wish List

- Better IDE (Smalltalk)
- Untyped language (SuperCollider)
  - Slang?
  - Scripting languages?
- Otherwise, just like CSL3!

Next Steps

- Unified buffer class (?)
- Better buffer usage
- New synthesis techniques (MAT 240D '03: granular synthesis and physical modeling)
- Tighter integration with CRAM
- Siren OSC routing in CSL server farms

Managing Siren and CSL: CRAM

- **CREATE Real-time Applications Manager (CRAM) = YADPE: Yet another distributed processing environment (DPE literature)**
- **Framework to deploy, start/stop, and monitor distributed real-time software**
- **Provides fault-tolerance and load-balancing***
- **Third implementation of a DPE for CREATE's distributed applications (HPDM, Yellow)**
- **V3: designed for simplicity and low overhead**

Components of a DPE

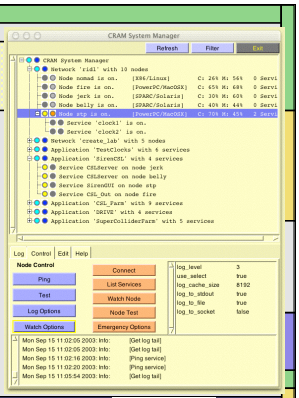
- Node manager program (Node)
  - Daemon that provides "remote exec" functionality and some status monitoring
- System manager (Manager)
  - Talks to nodes to run distributed apps
  - Uses DBs for network, services, and applications
  - May offer an ES-based planner
- Service interface (Service)
  - Wrapper code that is attached to server programs managed by the DPE

CRAM Operation Model

- Applications are composed of services
  - which probably talk among themselves unbeknownst to the management system
- Any program can be a service
- Nodes can run services
  - and monitor their hardware and services
- The manager talks to nodes to manage services
  - Start/stop, load-bal, fault-detection, etc.

CRAM Manager

- Network/Node
- Node/Service
- Application/Service
- Log/control pane



Using CRAM

- Write a Service class for the app
- It handles main() (creates the service object) and start/stop messages (start/stop the base app)
- Possibly implement run-time options
- Create a ServceType in the DB
- Add applications to the DB
- Go!

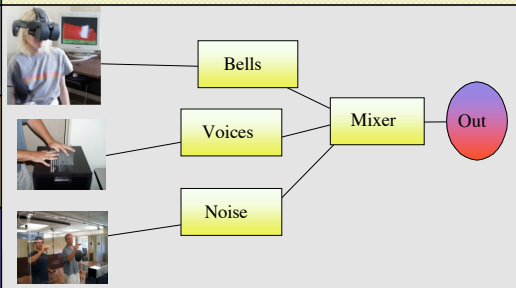
CRAM Implementation Details

- Node and Service interface written in C++ (~4kLOC)
- Manager written in Smalltalk (~4kLOC [largely auto-gen])
- Uses low-level UDP-based protocol
- Several levels of failure recovery, node/service discovery
- Flexible logging and monitoring

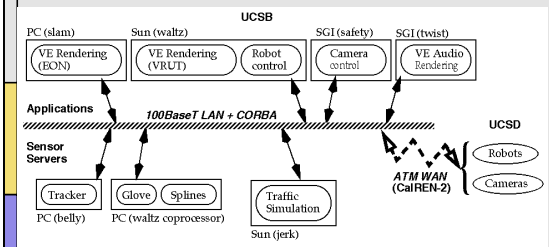
CRAM Databases

- Networks
  - Nodes and their properties
    - HW, OS, MIPS/MFLOPS, special I/O
- ServiceTypes
  - Name, arguments, options
  - HW/OS/IO requirements
- Services
  - Type and actual arguments
- Applications
  - Lists of services (& options) on nodes

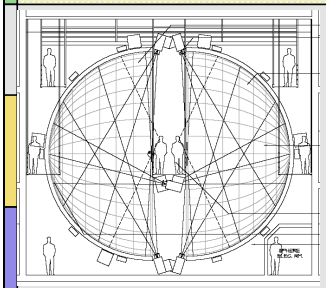
CRAM Configuration for CSL



ATON Distributed VE (2000)

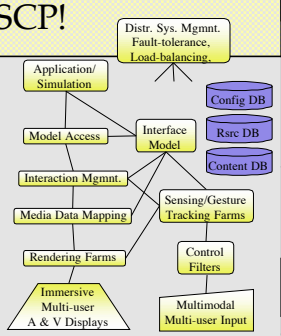


Why Siren/CSL/CRAM?



Why CRAM? DSCP!

- DSCP = MVC on steroids
- "Back-end" application models are science/numerical/simulation
- Presentation/interaction via CNSI Sphere, wide-area streaming
- Multimodal multi-user control and multimedia sensing
- Interface model = output data mappings + sensing/tracking policies
- Infrastructure uses distributed application management system
- Uses OO databases for configurations, resources (mgmnt. system), and media content (renderers)



"Challenges"

- True fault tolerance using off-the-shelf HW and OS
  - Robust node manager (monitoring/restart) (emergency thread)
  - Debugging (LA) network problems
- Performance and scalability issues
  - Node monitoring
  - Logging
  - Service creation

Conclusions

- For our requirements, we really had to start from scratch
- 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 fun of it!)
- See create.ucsb.edu/{Siren, CSL, CRAM}

Recent Pieces

- *Sensing/Speaking Space*
- *Four Magic Sentences*
- *Gates Still Open*
- *Eternal Dream (excerpt: finale movement)*
- *Leur Songe de la Paix*

Demo

- Siren GUI, tools, representation, composition code
- CSL examples
- EMA
- CRAM manager
- CRAM applications