The Siren Workbook

Outline

Introduction to the Siren 7.5 Workbook

Welcome to Siren 7.5!

What is this?

This document is the Siren Version 7.5 user's guide. This manual is available as a web site for on-line browsing (see http://FASTLabInc.com/Siren), as a PDF file for printing, or from within the Smalltalk run-time system in a workspace outline view (the best way to read it).

Throughout this document, there are executable Smalltalk expressions, which are generally enclosed in square brackets, as in

```
[ some code ]
```

If you are reading this text within Smalltalk, you can select this text by double-clicking on either of the square brackets, and then using the pop-up contextual menu to evaluate the expression (see the menu items "do it," "print it," "inspect it," or "debug it"). There is a more complete description of how to use these features in the section below on getting started.

Who is this intended for?

Before we start, readers should be reminded that Siren is a Smalltalk-80-based programming framework for developing music/sound applications. Siren is not an application in itself. The ideal user will be literate in some object-oriented

programming language, and possess at least a moderate understanding of musical terms.

For those unfamiliar with the syntax of Smalltalk, there is a section at the end of this outline that introduces the language.

How do I use this tool?

If you're reading this within Siren, the window is a workspace-list view. You can select different pages of the outline from the list above, and scroll within a page in this lower view. You can move the horizontal divider line (above) if you want a smaller topic list and larger text view.

Stephen Travis Pope (stp@HeavenEverywhere.com), Santa Barbara, April 2007

What is Siren?

The Siren system is a general-purpose software framework for music and sound composition, processing, performance, and analysis; it is a collection of about 375 classes written in Smalltalk-80. This version of Siren (7.5) works on VisualWorks Smalltalk (available for free for non-commercial use) and supports streaming I/O via OpenSoundControl (OSC), MIDI, and multi-channel audio ports. The Siren release is available via the web from the URL **http://FASTLabInc.com/Siren**. Note that you need a Smalltalk virtual machine and run-time to use Siren; you can down-load the free (non-commercial) system from Cincom at **http://www.cincom.com/smalltalk** or http://smalltalk.cincom.com/downloads/index.ssp

Siren is a programming framework and tool kit; the intended audience is Smalltalk developers, or users willing to learn Smalltalk in order to write their own applications. The built-in applications are meant as demonstrations of the use of the libraries, rather than as end-user applications. Siren is not a MIDI sequencer, nor a score notation editor, through both of these applications would be easy to implement with the Siren framework.

There are several elements to Siren:
- the Smoke music representation language
  (music magnitudes, events, event lists, generators, functions, and sounds);
- voices, schedulers and I/O drivers
  (real-time and file-based voices, sound, score file, OSC, and MIDI I/O);
- user interface components for musical applications
  (UI framework, tools, and widgets);
- several built-in applications
  (editors and browsers for Smoke objects); and
- external library interfaces for streaming I/O and DSP math
  (sound/MIDI I/O, fast FFT, CSL & Loris sound analysis/resynthesis packages)

Each of these components is described below in its own section of this document.

If you can read a bit of Smalltalk and want a quick tour before proceeding, read the condensed "Standard Siren Demo" that's at the end of this outline.

Where's More Documentation?

Siren and its predecessors and components (ARA, DoubleTalk, HyperScore ToolKit, and MODE) are documented in several extended book chapters and articles:
- "Squeak: Open Personal Computing and Multimedia" (Mark Guzdial and Kim Rose, eds, Prentice-Hall, 2002);
- "Musical Signal Processing" (C. Roads, S. T. Pope, G. DePoli, and A. Piccialli, eds. Swets & Zeitlinger, 1997);
- "The Interim DynaPiano" in "Computer Music Journal" 16:3, Fall, 1992 (also on the CMJ Web site);
- "The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology" (S. T. Pope, ed. MIT Press, 1991);
- Proceedings of the 1986, 1987, 1989, 1991, 1992, 1994, 1996, 1997, 2003 International Computer Music Conferences (ICMCs); and

There are more MODE- and Smoke-related documents (including the above references) in the directory ftp://FASTLabInc.com/Siren/Doc or as PDF files on the page http://HeavenEverywhere.com/stp/publs.html.

The official Siren home page is http://FASTLabInc.com/Siren.

Here are the on-line Docs: The best in-depth doc (book chapter) is in,
http://FASTLabInc.com/Siren/Doc/SirenBookChapter.pdf

The read the demo code workbook (this text), go to,
http://FASTLabInc.com/Siren/Siren7.5.Workbook.html
http://FASTLabInc.com/Siren/Siren7.5.Workbook.pdf

If you like to read manuals, take a look at,
http://FASTLabInc.com/Siren/Manual

An email discussion list called Siren is available; see the web page http://www.create.ucsb.edu/mailman/listinfo/Siren to sign up or to read the archives.

History

Siren and its predecessors stem from music systems that I've developed in the process of composing and realizing my music. Of the early ancestors, the MShell (1980-83) was the score processing shell used for "4" (1980-82); ARA (1982-4) was an outgrowth of the Lisp system used for "Bat out of Hell" (1983); the DoubleTalk system (1984-7) was based on the Smalltalk-80-based Petri net editing system used for "Requiem Aeternam dona Eis" (1986); the HyperScore ToolKit's various versions (1986-90) were used (among others) for "Day" (1988), and the MODE (1990-96) was developed to realize "Kombination XI" (1990) and "Paragraph 31: All Gates are Open" (1993).

Siren-on-Squeak (1996-2002) was a simple re-implementation of the MODE in the Squeak version of Smalltalk; it added the representations and tools I needed for "Four Magic Sentences" (1998-2000). Siren 7.4 added tools from the realizations of "Eternal Dream" (2002) and "Leur Songe de la Paix" (2003). The newest release incorporates new code from "Jerusalem's Secrets" and "Ora penso invece che il mondo..." (2005-6). In each of these cases, some amount of effort was spent--after the completion of the composition--to make the tools more general-purpose.

Portability

The Smalltalk portion of Siren is 100% cross-platform, and the DLLCC external interfaces to sound file, MIDI, and sound streaming I/O use cross-platform libraries. There are a few places (Sound play command and aubio interfaces) that assume UNIX shell commands can be run form within Smalltalk; I'm not certain hiow these port to Windows.

The core of Siren is not dependent on the dialect of Smalltalk, and in fact, runs well in Squeak. The bulk of the interactive tools and GUIs are based on my own display list graphics framework, and are thus also portable. The actual integrated applications and GUIs in Siren 7.5 use VisualWorks-specific application model classes, though.

Getting Started with Siren

Depending on what your background, you may take any one of several paths to get started with Siren.

RTFM: Please read the in-depth book chapter,
http://FASTLabInc.com/Siren/Doc/SirenBookChapter.pdf

If you're impatient to see what Siren can do, jump ahead to the demo script near the end of this outline.

If you're a literate programmer, but unfamiliar with the Smalltalk language or environment, skip ahead to the short introduction to reading the Smalltalk language at the end of this document.

If you're an experienced Smalltalk developer, I encourage you to read through the rest of this document, and use the in-line code examples as starting points to explore the implementation classes. Use the pop-up menu item "debug it" to get right into the code.

Executing code: do-it, print-it, inspect-it and debug-it

As a very simple introduction, look at the following line of code
```
3 + 4
```
this is a perfectly well-formed Smalltalk program. To execute it, you simply have to select the text, and then use the pop-up contextual menu to compile and run it; we call this "do-it" in the menu. Try selecting the above line and selecting the menu item "do-it."

You might notice that nothing visible happened. The system compiled and executed the program, but the threw away the result. To print the result object from an expression, use the menu item "print-it." Try this again with the above example code. This should now have printed the string "7" at the end of the text you selected.

To make text selection easier in this outline, I generally enclose text that's intended as a code example in square brackets (to make selecting the code easier -- you simply need to click on the opening or closing brackets). In many cases, I also place a "d,", "p," "i," or "db" after the text to give you a hint as to whether to do-it, print-it, inspect-it, or debug-it. Try this on the examples below.

```
[ 3 + 4 ] d
[ 3 + 4 ] p
```

```
[ 100 factorial ] p
[ 100 factorial ] i
```

Now try looking at the code of a Siren event list example; say "debug-it" to the expression below, which will bring up the code in a debugger; in this view, use the top-left menu bar button to "step-into" the block; now you'll be looking at the randomExample: method itself. You can use the 2nd button from the left to single step through this method, inspecting the objects in use with the inspectors at the bottom of the debugger.

```
[ EventList randomExample: 20 ] db
```

Setting up Siren

To test the Siren set-up and I/O, open the configuration/test panel,

```
[SirenSession openUtility] d
```

You can see the system configuration and test the MIDI, sound file, and sound IO here. If you use MIDI, use the left row of buttons to set it up and test it. The 2nd row of buttons is used to test sound IO via PortAudio. The SirenUtility is available as a menu item and a toolbar button in the Launcher's Tools menu (right-most button). There's a section below in this workbook about the Siren utility and transport panes.

The right-most buttons are important utilities. The "clean up" button stops the scheduler and shuts down MIDI and sound IO (for use in an emergency).

Special init

See the method SirenUtility class initialize, with which you can customize Siren so that it finds your devices, servers, and data files. See especially the lines,

```
DefaultMIDIIn := 4.              "Tune these to your setup"
DefaultMIDIOut := 10.
DefaultOSCHost := #[127 0 0 1].
DefaultOSCPort := 54321.
Verbosity := 1.            "0 = pretty silent, only error logging;
                    1 = medium-verbose IO interface logging;
                    2 = full scheduler and verbose interface logging"
[...]
                            "Set search paths"
self addDir: 'Databases' to: SoundDir.          "STP-specific"
self addDir: '3-Credo/*' to: SoundDir.          "You can add using wildcards"
```

Setting up the External Interfaces

Siren uses several external interfaces (based on the DLLCC framework or SWIG) for access to external data and I/O. The Smalltalk code for these interfaces is in the category MusicIO-External. The external libraries Siren uses are:

```
Sound file read/write -- libSndFile -- see http://www.mega-nerd.com/libsndfile
Streaming MIDI I/O -- PortMidi -- see http://www.cs.cmu.edu/~music/portmusic
Streaming sound I/O -- PortAudio -- see http://www.portaudio.com
Fast Fourier Transform -- FFTW -- see http://fftw.org
```

You need to have these libraries installed (normally in a directory such as /usr/local/lib), and compile and link the C-language interface libraries in the subdirectory DLLCC for sound or MIDI I/O to work with Siren. Binaries are available for Mac OSX. There are full sources and pre-compiled versions of the required libraries on the CSL web site; look at http://FASTLabInc.com/CSL.

To get Loris, http://sourceforge.net/projects/loris
To get CSL, http://FASTLabInc.com/CSL

To use the SWIG-based interfaces (Loris and CSL), see the pages below.

Testing MIDI I/O
This will open the MIDI driver and play a note; printing messages to the Transcript.

```
[MIDIPort testANote]
```

Test this using a MIDI dump utility (such as MIDIMonitor), or a soft-synth, or outboard MIDI hardware. There are many other tests in class MIDIPort.

Testing OpenSoundControl output
The following example will send out a few OSC messages; test it with a dumpOSC utility, or with the CSL OSC server example. There are many other tests in class OSCVoice.

[OSCVoice fm_example1]

Testing sound file I/O
This will print diagnostics and a few rows of sample values to the Transcript. There are more examples ion class SoundFile.
[LibSndFileInterface example1: 'a.snd']

Testing streaming sound I/O
This will report to the Transcript; lsee also the examples in class SoundPort.
[PortAudioInterface example1]

Testing the Siren scheduler
To test the built-in real-time scheduler, try the following block, which will display colored rectangles on the top window for 5 seconds, then refresh the screen.
[ActionEvent playExample]

MIDI/Sound Configuration

Siren's MIDI support depends on a platform-independent interface class that talks to VM-side primitives that talk to OS-level device drivers. There are clas ssettings in MIDIPort for the default I/O devices. The same applies to SoundPort.

Smalltalk Options

There are several configurable parts to Siren. Class SirenUtility is the general place to find utility messages related to Siren set-up and global variables. Look at its class variables and initialization method.

Several of the voice classes have "default" methods that return a default instance. Look at MIDIVoice default, and Voice default.

Siren looks in default directory for scores and sound files. By default this is called "Data" and is a sub-folder of the folder where the virtual image is executing. There are methods in class SirenUtility to change this.

Useful Utilities

There are several sound/MIDI utilities that Siren users generally need; these include:
a MIDI dump utility such as MIDIMonitor from Snoize;
an OSC dump utility such as dumpOSC from CNMAT;
an OSC send script such as sendOSC from CNMAT; and
an audio patching utility such as Jack or Audio Hijack.

Siren House-keeping

To clear out temp. event lists, use,
[SirenSession flushTempEventLists]
or to flush all,
[SirenSession flushAllEventLists]

To flush and close down the scheduler,
[Schedule interrupt; flush; release]

To send MIDI all notes off, flush ports, throw away open ports, clear out temp event lists, etc.
[MIDIPort cleanUp]

Check here to see if there are any left-over objecgts being held onto,
[DependentsFields inspect]

Siren Design Notes

There are several elements to Siren:
the Smoke music representation language
(music magnitudes, events, event lists, generators, functions, and sounds);
voices, schedulers and I/O drivers
(real-time and file-based voices, sound and MIDI I/O);
user interface components for musical applications
(extended graphics framework, layout managers, UI tools and widgets); and
several built-in applications
(editors and browsers for Siren objects).

Siren Design Patterns

To understand Siren's design, you should be familiar with the terminology of object-oriented design patterns, as laid out in the books "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma, Helm, Johnson, and Vlissides, Addison-Wesley 1995), and "The Design Patterns Smalltalk Companion" (Alpert, Brown, and Woolf, Addison-Wesley 1998).

The following list introduces the design patterns found in Siren, and gives the parts of the system where they are used and a brief definition of each.

Composite -- events/event lists, display items/display lists -- class structure for building hierarchies of objects

Adaptor -- voices, ports, graphics -- interface objects translate between message protocol "languages"

Singleton -- ports, scheduler, external interfaces -- a class is limited to having a single well-known instance.

Decorator -- event modifiers, voices, layout -- one object "wraps" another and forwards messages sent to it.

Observer -- MVC, MIDI -- an object registers istelf as an "observer" of some aspect of another object, wanting to get update messages when the observed object changes.

Strategy -- layout managers, event generators, voices -- a set of classes provide a family of algorithms that encapsulate their client objects.

Proxy -- voices, ports, Smoke -- one object serves as a representative of another in some context.

Chain of Responsibility -- MVC, voices, input -- object-oriented recursion iterates through tree structures using composed command objects.

Visitor -- Smoke, voices, graphics -- active objects traverse data structures operating on them.

Double-dispatching -- Smoke -- polymorphic operators support mixed-mode operations among families of classes.

Multi-threading -- scheduler

The Siren Class Categories

The list below is the class categories (akin to subpackages) in the Siren namespace. The same list is a class message (categoryList) in class SirenUtility.

Music-Models-Representation -- abstract music magnitude models
Music-Models-Implementation -- music magnitude implementation classes
Music-Events -- central event classes, EventLists
Music-EventGenerators -- EventGenerators: chords, clouds, etc.
Music-EventModifiers -- EventModifiers: rubato, swell
Music-Functions -- functions of time
Music-Sound -- abstract and concrete sound classes
Music-Support -- Scheduler, Utility, Session classes, abstract device, port and editor models
Music-PitchClasses -- Pitch classes (octave-independent pitch notation), chords and key signatures
Music-PitchScales -- Scales and harmony

MusicIO-External -- External interfaces to PortAudio, PortMIDI, LibSndFile, etc.
MusicIO-MIDI -- MIDI voices and devices
MusicIO-OSC -- OSC streams
MusicIO-Sound -- Sound ports
MusicIO-Voices -- Voice hierarchy

MusicUI-DisplayLists -- Core display items and list
MusicUI-DisplayListViews -- Basic DL views
MusicUI-Editors -- Music editors and support
MusicUI-Functions -- Function editors
MusicUI-Layout -- Layout managers for various notations
MusicUI-Sound -- Sound and spectrum views

MusicApps-CSL -- Support for the CREATE Signal Library (CSL)
MusicApps-Loris -- Support for the Loris analysis/resynthesis package
MusicApps-LPC -- Rudiomentary support for linear predictive coding
MusicApps-SHARC -- Readers for the Sandell Harmonic Archive timbre database

There are separate sub-namespaces for the SWIG-generated CSL and Loris classes.

The primary class hierarchies of Siren are given below grouped into categories. The text indentation signifies sub/super-class relationships, and instances variable names are shown.

Music Magnitude Models

```
Magnitude
  MusicMagnitude -- value
    MusicModel -- class generality table
      Chroma
        ModeMember
          Pitch
      Chronos
        Duration
        Meter
      Ergon
        Amplitude
      Positus
        Directionality
        Position
        Spatialization
```

Music Magnitude Implementations

```
Magnitude
  MusicMagnitude -- value
    ConditionalDuration
    NominalMagnitude
      SymbolicLoudness
      SymbolicPitch
    NumericalMagnitude
      HertzPitch
      IntervalMagnitude
        MIDIPitch
        MIDIVelocity
      MSecondDuration
        SecondDuration
      RatioMagnitude -- relative
        RatioDuration
        RatioLoudness
          DBLoudness
        RatioPitch
    OrdinalMagnitude -- table
      Length
      Sharpness
    PField -- name field
```

Events

```
AbstractEvent -- properties
  DurationEvent -- duration
    ActionEvent -- action
    MusicEvent -- pitch loudness voice
      EventList -- events index startedAt
```

EventLists

```
AbstractEvent -- properties
  DurationEvent -- duration
    MusicEvent -- pitch loudness voice
      EventList -- events index startedAt
        EventGenerator
          Cloud -- density
            DynamicCloud
            SelectionCloud
              DynamicSelectionCloud
          Cluster
            Chord -- root inversion
              Arpeggio -- delay
            Roll -- number delta noteDuration
              Trill
          Ostinato -- list playing process
```

Functions

```
AbstractEvent -- properties
  DurationEvent -- duration
    Function -- data range domain
      FourierSummation -- myForm myArray
      LinearFunction
        ExponentialFunction
        SplineFunction -- linSeg
      Sound
        GranularSound -- grains
        StoredSound -- samplesInMemory firstIndex changed
          FloatSound
          VirtualSound -- source
            CompositeSound -- components
            GapSound -- cutList
          WordSound
```

Voices

```
Model -- dependents
  Voice -- name instrument stream
    MIDIFileVoice -- fileType tracks ppq tempo
    MIDIVoice -- currentTime
    NotelistVoice -- parameterMap
      CmixVoice
      CmusicVoice
      CsoundVoice
    SoundVoice
```

The Smoke Music Representation

The "kernel" of Siren is in the classes related to representing the basic musical magnitudes (pitch, loudness, duration, etc.), and for creating and manipulating event and event list objects. This package is known as the Smallmusic Object Kernel (Smoke--name suggested by Danny Oppenheim).

Smoke is an implementation-language-independent music representation, description language, and interchange format that was developed by a group of researchers at CCRMA/Stanford and CNMAT/Berkeley during 1990/91.

The basic design requirements are that the representation support the following:
--abstract models of the basic musical quantities (scalar magnitudes such as pitch, loudness, and duration);
--flexible grain-size of "events" in terms of "notes," "grains," "elements," or "textures";
--event, control, and sampled sound processing description levels;

--nested/hierarchical event-tree structures for flexible description of "parts," "tracks," or other parallel or sequential organizations;
--annotation and marking of event tree structures supporting the creation of heterarchies (lattices) and hypermedia networks;
--annotation including common-practise notation possible;

--instrument/note (voice/event, performer/music) abstractions;
--separation of "data" from "interpretation" ("what" vs. "how" in terms of providing for interpretation objects--voices);

--abstractions for the description of "middle-level" musical structures (e.g., chords, clusters, or trills);
--sound functions, granular description, or other (non-note-oriented) description abstractions;
--description of sampled sound synthesis and processing models such as sound file mixing or DSP;

--possibility of building convertors for many common formats, such as MIDI data, Adagio, note lists, DSP code, instrument definitions, or mixing scripts; and
--possibility of parsing live performance into some rendition in the representation, and of interpreting it (in some rendition) in real-time.

The "executive summary" of Smoke from (from the 1992 ICMC paper) is as follows. Music (i.e., a musical surface or structure), can be represented as a series of "events" (which generally last from tens of msec to tens of sec). Events are simply property lists or dictionaries that are defined for some duration; they can have named properties whose values are arbitrary. These properties may be music-specific objects (such as pitches or spatial positions), and models of many common musical magnitudes are provided.

Events are grouped into "event lists" (AKA composite events or event collections) by their relative start times. Event lists are events themselves and can therefore be nested into trees (i.e., an event list can have another event list as one of its events); they can also map their properties onto their component events. This means that an event can be "shared" by being in more than one event list at different relative start times and with different properties mapped onto

it.

Events and event lists are "performed" by the action of a scheduler passing them to an interpretation object or voice. Voice objects and applications determine the interpretation of events' properties, and may assume the existence of "standard" property names such as pitch, loudness, voice, duration, or position. Voices map application-independent event properties onto the specific parameters of I/O devices or formatted files. A scheduler expands and/or maps event lists and sends their events to their voices in real time.

Sampled sounds can also be described as objects, by means of synthesis "patches," or signal processing scripts involving a vocabulary of sound manipulation messages.

Examples

Move to the following sections for extensive examples of Smoke object creation and manipulation.

Smoke Music Magnitude Models

Smoke uses objects called music magnitudes to represent the basic "units of measure" of musical sound: duration, pitch, amplitude, etc. MusicMagnitude objects are characterized by their identity, class, species, and value (e.g., the pitch object that represents 'c3' has its object identity, the class SymbolicPitch, the species Pitch, and the value 'c3' [a string]). MusicMagnitude behaviors distinguish between class membership and species in a multiple-inheritance-like scheme that allows the object representing "440.0 Hz" to have pitch-like and limited-precision-real-number-like behaviors. This means that its behavior can depend on what it represents (a pitch), or how its value is stored (a floating-point number).

The mixed-mode music magnitude arithmetic is defined using the technique of species-based coercion, i.e., class Pitch knows whether a note name or Hertz value is more general. This provides capabilities similar to those of systems that use the techniques of multiple inheritance and multiple polymorphism (such as C++ and the Common Lisp Object System), but in a much simpler and scalable manner. All meaningful coercion messages (e.g., (440.0 Hz asMIDIKeyNumber)), and mixed-mode operations (e.g., (1/4 beat + 80 msec)) are defined.

The basic model classes include Pitch, Loudness, and Duration; exemplary extensions include Length, Sharpness, Weight, and Breath for composition- or notation-specific magnitudes. The handling of time as a parameter is finessed via the abstraction of duration. All times are durations of events or delays, so that no "real" or "absolute" time object is needed. Duration objects can have simple numerical or symbolic values, or they can be conditions (e.g., the duration until some event X occurs), Boolean expressions of other durations, or arbitrary blocks of Smalltalk-80 code.

Functions of one or more variables are yet another type of signal-like music magnitude. The MODE Function class hierarchy includes (e.g.,) line segment, exponential segment, spline segment and Fourier summation functions.

In the verbose SmOKe format music magnitudes, events and event lists are created by instance creation messages sent to the appropriate classes. The first three expressions in the examples below create various music magnitudes and coerce them into other representations.

The terse form for music magnitude creation uses post-operators (unary messages) such as 440 hz or 250 msec, as shown in the examples below.

Users can extend the music magnitude framework with their own classes that refine the existing models of define totally new kinds of musical metrics.

Basic MusicMagnitude Models

Durations
  SecondDuration -- 1 sec
  MSecondDuration -- 100 msec (milliseconds)
  USecondDuration -- 100000 usec (microseconds)
  RatioDuration -- 1/4 beat (relative fractions of some tempo scale)
  ConditionalDuration -- until: [:t | boolean-block]

Pitches
  HertzPitch -- 440.0 hz
  MIDIPitch -- 60 pitch -- (or 60 key) can be non-integer for microtonal tunings (use #asFracMIDI)
  SymbolicPitch -- 'c#3' pitch -- several different pitch spellings supported
  RatioPitch -- 11/9 of: anotherPitch -- used for fraction-oriented tunings

Amplitude/Loudness Objects
  DBLoudness -- -3 dB -- can be negative relative to 0 dB or positive-valued
  RatioLoudness -- 0.7071 ampl (or 0.7071 loudness) -- range of 0.0 to 1.0
  SymbolicLoudness -- 'fff' ampl

MIDIVelocity -- 96 velocity

Other Music Magnitudes
OrdinalMagnitudes -- have order but no explicit value
PField -- name/slot/value -- used for note lists
See also magnitude accessors in LayoutManagers

MusicMagnitude Examples

Verbose MusicMagnitude Creation and Coercion Messages

```
(Duration value: 1/16) asMsec        "Answers Duration 62 msec."
(Pitch value: 60) asHertz            "Answers Pitch 261.623 Hz."
(Amplitude value: 'ff') asMIDI       "Answers MIDI key velocity 100."
```

Terse MusicMagnitude Creation using post-operators

```
440 Hz                "a HertzPitch"
'c#3' pitch           "a SymbolicPitch"
'cs3' pitch           "a SymbolicPitch"
60 pitch              "a MIDIPtch"
60 key                "a MIDIPtch"
250 msec              "a MSecondDuration"
1/4 beat              "a RatioDuration"
-16 dB                "a DBLoudness"
'fff' ampl            "a SymbolicLoudness"
0.1 ampl              "a RatioLoudness"
```

MusicMagnitude Coercion Examples

```
440 Hz asSymbol       "--> 'a3' pitch"
(1/4 beat) asMsec     "--> 250 msec"
#mf ampl asMIDI       "--> 70 vel"
-16 dB asRatio value  "--> 0.158489"
0.1 ampl asMIDI       "--> 12 vel"
```

Duration Coercion Example--create a 1/8 beat duration and coerce it into a couple of other representations, printing the result to the Smalltalk transcript. To execute this, double-click just inside the open-bracket to select the entire expression and use the pop-up menu or command key (control-d) to "do it."

```
[ | me |
me := Duration value: 1/8.
Transcript cr; show: me printString, ' = ',
   me asSec printString, ' = ',
   me asUsec printString; cr] d
```

Pitch Coercion Example--create a named pitch (middle C) and print it to the transcript as Hz and as a MIDI key number.

```
[ | me |
me := Pitch value: 'c3'.
Transcript show: me printString, ' = ',
   me asHertz printString, ' = ',
   me asMIDI printString; cr.
"me inspect"] d
```

Amplitude Coercion Example--create a named dynamic value and print it as an amplitude ratio and a MIDI velocity.

```
[ | me |
me := Amplitude value: #mf.
Transcript show: me printString, ' = ',
   me asRatio printString, ' = ',
   me asDB printString, ' = ',
   me asMIDI printString; cr.
"me inspect"] d
```

Mixed-mode Arithmetic--demonstrate adding beats and msec, or note names and Hertz values. Select and print these.

```
[(1/2 beat) + 100 msec]          " (0.6 beat")
['a4' pitch + 25 Hz]             " (465.0 Hz)"
[('a4' pitch + 100 Hz) asMIDI]   " (73 key)"
```

```
[('a4' pitch + 100 Hz) asFracMIDI]   " (72.5455 key)"
['mp' ampl + 3 dB]                   " (-4.6 dB)"
```

Microtonality

Alberto de Campo's microtonal extensions allow MIDI pitches to be floating-point numbers (e.g., MIDI key 60.25) and named pitches to have "remainder" values (e.g., c3 + 25 cents) as in the following examples.

```
[438 Hz asSymbol]          "rounds to nearest chromatic note, a3."
[443.5 Hz asMIDI]          "ditto."
[265 Hz asFracMIDI]        "converts to float chromatics; can be rounded, used
                          for MIDI pitch bend or for precise synthesis in Hz."
[61.26 key asHertz]        "float chromatics can also be used directly; for
                          microtonal scales this is clearer than Hz (to me at least)."
[260.0 Hz asFracSymbol]    "is rounded, but keeps track of offsets in
                           an inst var (fracPitch); survives conversions etc."
```

Note that asMIDI and asSymbol can now be used to round pitches to chromatics, while the messages asFracMIDI and asFracSymbol keep the full microtonal precision.

Conditional Durations

Conditional durations allow one to have state-dependent events lists.

```
[ConditionalDuration exampleWithRands]
[ConditionalDuration until: [ :x | x > 5]]
```

Extending Siren's MusicMagnitude Models

Developers and composers can extend the core Smoke representation by adding new magnitude models (abstract species classes with instance creation methods, post-fix operators and a generality table) and concrete magnitudes (with units, coercion methods, and print/store). This is relatively easy and can be very useful for a composition that uses a new model for its structure.

MusES Pitch Classes, Intervals, Scales, and Chords

Siren incorporates Francois Pachet's MusES pitch class package. A pitch class is an abstraction of a single pitch or note name (e.g., C in any octave, or 'fa' in any key), and there are relationships for pitch class spelling (f#, la flat), pitch equivalence, enharmonic equivalence (do sharp = re flat), and interval algebra.

Based on this, chords and scales can be defined according to pitch class rules.

To make the code easy to use, MusES puts a shortcut in the Smalltalk global dictionary; the global variable N holds the PitchClass class, making the following code examples easier to read.

Print these to see results, or look at the advanced EventGenerator examples.

Pitch expressions (print these)

```
[N C sharp sharp]
[N C augmentedFourth]
[N C diminishedThirteenth]
[N C augmentedFourth pitchEqual: N C diminishedFifth]
[N D sharp closestEnharmonic]
[N E flat closestEnharmonic]
```

Intervals

```
[N do flat intervalBetween: N sol]
[MusicalInterval perfectFourth topIfBottomIs: N F sharp]
[MusicalInterval allIntervalsType: 4]
```

Scales

```
[PitchClass B majorScale notes]
[PitchClass D flat melodicMinorScale notes]
```

To use a scale as real pitches, you need to give it an octave, as in

```
[PitchClass C majorScale asPitchesInOctave: 5]
[(HungarianMinor root: N fa) asPitchesInOctave: 2]
```

Chords

```
(PitchClassChord new fromString: 'C') notes
```

```
(PitchClassChord new fromString: 'Re maj7') notes
(PitchClassChord new fromString: 'C 13 aug9') notes
(PitchClassChord new fromString: 'C aug9 dim5') notes
```

Key signatures

```
[Signature choseSignature]
[(Signature new nbSharps: 4) tonality]
[Signature fromTonality: N E flat majorScale]
```

EventList Examples using Scales

```
[((NeapolitanMinor root: N do) generateChordsPoly: 5 inOctave: 3) scrambled] p
```

Extending the Models of Pitch Classes and Scales

Obviously, this framework is designed for ease of extension, and new interval types, scale rules, families of key signatures are all straightforward to add.

Smoke Events

All musical structures in Smoke--from micro-sound components of a note, to entire compositions--are represented as event objects. Events are very simple objects that have lists of properties, and get/set methods for managing these properties. The Event object in Smoke is modeled as a property-list dictionary with a duration. Events have no notion of external time until their durations become active. Event behaviors include duration and property accessing, and "performance," where the semantics of the operation depends on another object--a voice or driver as described below.

The event classes are quite simple; events have little interesting behavior (most of that being taken over by event lists and voices), and there is not a rich hierarchy of kinds of events.

The primary messages that events understand are property getter/setters,

```
(anEvent duration: someDurationObject)
```

to set the duration time of the event to some Duration-type music magnitude, and property accessing messages such as

```
(anEvent color: #blue)
```

to set the "color" (an arbitrary property) to an arbitrary value (the symbol #blue).

The meaning of an event's properties is interpreted by voices and user interface objects; it is obvious that (e.g.,) a pitch could be mapped differently by a MIDI output voice and a graphical notation editor. It is common to have events with complex objects as properties (e.g., envelope functions, real-time controller maps, DSP scripts, structural annotation, version history, or compositional algorithms), or with more than one copy of some properties (e.g., one event with enharmonic pitch name, key number, and frequency, each of which may be interpreted differently by various voices or structure accessors). The distinction between a complex event object model and adaptors and visitors with dynamic state is intentionally blurred.

That there is no prescribed "level" or "grain size" for events in Smoke. There may be a one-to-one or many-to-one relationship between events and "notes," or single event objects may be used to represent long complex textures or surfaces.

Note the way that Smoke uses the Smalltalk concatenation message "," (comma) to denote the construction of events and event lists; (magnitude, magnitude) means to build an event with the two magnitudes as properties, and (event, event) or ((duration -> event) , (duration -> event)) means to build an event list with the given events as components.

There are classes for events are as follows.

```
AbstractEvent -- just a property list
DurationEvent -- adds duration and scheduling behavior
MusicEvent -- adds pitch, loudness and voice
ActionEvent -- has a block that it evaluates when scheduled
```

It is seldom necessary to extend the hierarchy of events.

Event Creation Examples

Verbose Event Creation Messages -- Class messages

```
"Create a `generic' event."
   MusicEvent duration: 1/4 pitch: 'c3' ampl: 'mf'

"Create one with added properties."
   (MusicEvent dur: 1/4 pitch: 'c3') color: #green; accent: #sfz
```

Terse Event Creation using concatenation of music magnitudes--inspect these.

```
[440 Hz, (1/4 beat), 44 dB]

[490 Hz, (1/7 beat), 56 dB, (#voice -> #flute), (#embrochure -> #tight)]

[(#c4 pitch, 0.21 sec, 64 velocity) voice: MIDIVoice default]
```

Smoke Event Lists

In accordance to the "composite" OO design pattern, EventList objects hold onto collections of events that are tagged and sorted by their start times (represented as the duration between the start time of the container event list and that of the constituent event). The event list classes are subclasses of Event themselves. This means that event lists can behave like events and can therefore be arbitrarily deeply nested, i.e., one event list can contain another as one of its events.

The primary messages to which event lists respond (in addition to the behavior they inherit by being events), are
(anEventList add: anEvent at: aDuration) -- to add an event to the list
(anEventList play) -- to play the event list on its voice (or a default one)
(anEventList edit) -- to open a graphical editor in the event list
(anEventList open) -- either edit, play or inspect, depending on the and keys
and Smalltalk-80 collection iteration and enumeration messages such as
(anEventList do: [1-arg-block]) -- to iterate over a list's events
(anEventList select: [Bool-Block]) -- to select the events that satisfy the given (Boolean) function block.

Event lists can map their own properties onto their events in several ways. Properties can be defined as lazy or eager, to signify whether they map themselves when created (eagerly) or when the event list is performed (lazily). This makes it easy to create several event lists that have copies of the same events and map their own properties onto the events at performance time under interactive control. Voices handle mapping of event list properties via event modifiers, as described below.

In a typical hierarchical Smoke score, data structure composition is used to manage the large number of events, event generators and event modifiers necessary to describe a full performance. The score is a tree--possibly a forest (i.e., with multiple roots) or a lattice (i.e., with cross-branch links between the inner nodes)--of hierarchical event lists representing sections, parts, tracks, phrases, chords, or whatever abstractions the user desires to define. Smoke does not define any fixed event list subclasses for these types; they are all various compositions of parallel or sequential event lists.

Note that events do not know their start times; this is always relative to some outer scope. This means that events can be shared among many event lists, the extreme case being an entire composition where one event is shared and mapped by many different event lists (as described in [Scaletti 1989]). The fact that the Smoke text-based event and event list description format consists of executable Smalltalk-80 message expressions (see examples below), means that it can be seen as either a declarative or a procedural description language. The goal is to provide "something of a cross between a music notation and a programming language" (Dannenberg 1989).

Event List Examples

The verbose way of creating an event list is to create a named instance and add events explicitly as shown in the first example below, which creates a D-major chord.

```
[((EventList newNamed: #Chord1)
    add: (1/2 beat, 'd3' pitch, 'mf' ampl) at: 0;
    add: (1/2 beat, 'fs3' pitch, 'mf' ampl) at: 0;
    add: (1/2 beat, 'a4' pitch, 'mf' ampl) at: 0) open]
```

This same chord could be defined more tersely as a dictionary of (duration => event) pairs,

```
[((0 => (1/2 beat, 'd3' pitch, 'mf' ampl)),
 (0 => (1/2 beat, 'fs3' pitch, 'mf' ampl)),
 (0 => (1/2 beat, 'a4' pitch, 'mf' ampl))) open]
```

Note the use of the "=>" message, which works just like Smalltalk's "->'" in that it creates an association between the key on the left and the value on the right; the difference is that it creates a special kind of association called an EventAssociation.

This could be done even more compactly using a Chord object (see the discussion of event generators below) as,

```
[(Chord majorTriadOn: 'd3' inversion: 0) eventList open]
```

Terse EventList creation using concatenation of events or (duration, event) asociations looks like,

```
[(440 Hz, (1/2 beat), 44.7 dB),       "note the comma between events"
 (1 => ((1.396 sec, 0.714 ampl) phoneme: #xu))]   "2nd event starts at 1 second"
```

Bach Example--First measure of Fugue 2 from the Well-Tempered Klavier (ignoring the initial rest).

```
    ((0 beat) => (1/16 beat, 'c3' pitch)),
    ((1/16 beat) => (1/16 beat, 'b2' pitch)),
    ((1/8 beat) => (1/8 beat, 'c3' pitch)),
    ((1/4 beat) => (1/8 beat, 'g2' pitch)),
    ((3/8 beat) => (1/8 beat, 'a-flat2' pitch)),
    ((1/2 beat) => (1/16 beat, 'c3' pitch)),
    ((1/16 beat) => (1/16 beat, 'b2' pitch)),
    ((1/8 beat) => (1/8 beat, 'c3' pitch)),
    ((3/4 beat) => (1/8 beat, 'd3' pitch)),
    ((7/8 beat) => (1/8 beat, 'g2' pitch))
```

There are more comfortable event list creation methods, such as the following examples.

Play a chromatic scale giving the initial and final pitches (as MIDI key numbers) and total duration (in seconds)

```
[(EventList scaleFrom: 48 to: 60 in: 1.5) open] d
```

Create 64 random events with parameters in the given ranges and play them over the default output voice.

```
[(EventList randomExample: 32
   from: ((#duration: -> (50 to: 200)),       "durations in msec"
         (#pitch: -> (36 to: 60)),          "pitchs as MIDI key numbers"
         (#ampl: -> (48 to: 120)),       "amplitudes as MIDI key velocities"
         (#voice: -> (1 to: 1)))) open]    "voices as numbers"
```

Note that the argument for the keyword "from:" is a dictionary in the form (property-name -> value-interval).

Same with named instruments = play using named instruments

```
[(EventList randomExample: 64
   from: ((#duration: -> (0.15 to: 0.4)),         "dur in sec"
         (#pitch: -> (36 to: 60)),              "MIDI pitch"
         (#ampl: -> (48 to: 120)),            "MIDI velocity"
         (#voice: -> #(organ1 flute2 clarinet bassoon1 marimba bass1)))) open]
```

Event lists don't have to have pitches at all, as in the word,

```
[EventList named: 'phrase1'
   fromSelectors: #(duration: loudness: phoneme:)         "3 parameters"
   values: (Array with: #(595 545 545 540 570 800 540)    "3 value arrays"
            with: #(0.8 0.4 0.5 0.3 0.2 0.7 0.1)
           with: #(#dun #kel #kam #mer #ge #sprae #che)).
(EventList named: 'phrase1') inspect]
```

Note the format of the arguments to the message "fromSelectors: values:" used above, the first is an array of property selector symbols, and the second is an array of arrays for the property data

There are a number of useful EventList instance creation methods based on the "fromSelectors: values:" method, for example, the following example shows the use of data lists in a serial style for score creation.

```
[(EventList serialExample: 64
   from: ((#duration: -> #(0.1 0.1 0.1 0.2)), (#pitch: -> #(48 50 52 53 52)),
        (#ampl: -> #(48 64)), (#voice: -> #(1)))) open]
```

This example creates a scale where the event property types (duration, pitch, amplitude) are mixed.

```
[EventList scaleExample2] i
```

Here's another example of creating a simple melody

```
[(EventList named: 'melody'
    fromSelectors: #(pitch: duration:)
   values: (Array with: #(c d e f g)
            with: #(4 8 8 4 4) reciprocal)) open]
```

You can create event lists with snippets of code such as the following whole-tone scale.

```
[ | elist |
elist := EventList newAnonymous.
1 to: 12 do:
   [ :index |
   elist add: (1/4 beat, (index * 2 + 36) key, #mf ampl)].
elist open ]
```

Event lists can be nested into arbitrary structures, as in the following group of four sub-groups

```
[(EventList newNamed: 'Hierarchical/4Groups')
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (36 to: 40)), (#ampl: -> #(110)))) at: 0;
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (40 to: 44)), (#ampl: -> #(100)))) at: 1;
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (44 to: 48)), (#ampl: -> #(80)))) at: 2;
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (48 to: 52)), (#ampl: -> #(70)))) at: 3;
   open "inspect" ]
```

Smalltalk methods can process event lists in many different ways, as in this code that uses the eventsDo: [] message to increase the durations of the last notes in each of the subgroups from the previous example.

```
[(EventList named: 'Hierarchical/4Groups') eventsDo:
   [ :sublist | | evnt |          "Remember: this is hierarchical, to the events are the sub-groups"
   evnt := sublist events last event.       "get the first note of each group"
   evnt duration: evnt duration * 4].       "multiply the duration by 4"
(EventList named: #groups) open ]
```

In the following example, we iterate over a scale and make it slow down to changing the event start times

```
[ | elist |
elist := EventList scaleExampleFrom: 60 to: 36 in: 3.
1 to: elist size do:
   [ :index | | assoc |
   assoc := elist events at: index.
   assoc key: (assoc key * (1 + (index / elist events size)))].
elist open ]
```

There are many more event list processing examples in the various class example methods, and in the event generator example that follow in this workbook.

Storage and Utilities

Note the use of event list names in the above examples. All named event lists are stored in a hierarchical dictionary named EventLists that's held in class SirenSession. To look at all named event lists, execute the following

```
[SirenSession eventLists] i
```

If you create an event list with a name that contains the character '/', then it is assumed to be in a subdictionary of the top-level event list dictionary, as in the example above that created an event list named 'Hierarchical/4Groups.' You can use this to manage your own sketches and pieces. If you create an event list named 'Opus1/Prelude/Exposition/Theme1' then the hierarchy of implicit in the name will be reflected by an automatically created hierarchical set of event list dictionaries.

```
"SirenSession eventList: 'piece1/mvmnt1/part1' put: EventList new"
"SirenSession eventList: 'piece1/mvmnt1/part1'"
```

There's a pop-up menu in the transport view that allows you to select event lists from this hierarchy.

You can erase the temporary lists (those in the dictionary named #Temp) from the EventList dictionary with,

```
[SirenSession flushTempEventLists]
```

or to flush all,

```
[SirenSession flushAllEventLists]
```

Inspect a dictionary of all known event lists.

```
[SirenSession eventLists inspect]
```

To read in a stored file, simply,

```
[(Filename named: 'events.st') fileIn]
```

Load all event lists (.ev, .midi, and .gio files), from the data directories.

```
[SirenSession loadDemoData]
```

Extending The Event/EventList Framework

There are a number of approaches one can take to extend the Smoke framework. The core event classes rarely need extension of subclassing. Simple methods in a workspace or class example methods can create and process event lists through many stages, and the built-in persistency and versioning in s7 files aids composers in the stages of content creation, refinement, sorting, and mixing.

It is often useful to add new EventList instance creation method for short-hand representations of chords, scales, etc. One can also write new processing methods (e.g., filters) in the EventList class. These can also be placed on EventGenerator or EventModifier classes (see the next workbook sections).

Siren Functions

There are several classes that represent functions of 1 variable (typ. time) such as envelopes or waveforms. These objects can be described using (e.g.,) linear or exponential interpolation between n-dimensional break-points, Fourier sine summation, cubic splines, or as raw sampled data. In addition to the instance creation methods, Functions understand array-like accessing method at: or atX: to get at their values.

The FunctionView displays up to 4 functions and supports simple editing.

Function Creation Examples

The expressions below demonstrate the various creation techniques for Siren functions.

If you like thicker function plots, do this,

```
[FunctionView lineWidth: 2]
```

Basic ramp up/down (linear and exponential flavors)

```
[(LinearFunction from: #((0 0) (0.5 1) (1 0))) at: 0.25] p
[(ExponentialFunction from: #((0 0 5) (0.5 1 -5) (1 0))) at: 0.25 ] p
```

ADSR-like envelopes

```
[(LinearFunction from: #((0 0) (0.1 1) (0.16 0.7) (0.8 0.4) (1 0))) edit]
[(ExponentialFunction from: #((0 0 5) (0.1 1 -3) (0.8 0.4 -2) (1 0))) edit]
```

Open a view with a linear envelope, an exponential envelope, a spline curve, and a sum-of-sines function

```
[FunctionView multiFunctionExample]
```

Sine Summation

```
[(FourierSummation from: #((1 1 0) (3 0.3 0) (5 0.2 0) (7 0.15 0) (9 0.11 0) (11 0.09 0))) edit]
```

Others

```
[(Function randomOfSize: 512 from: 0.2 to: 0.9) edit]
[FunctionView onFunction:
   (Function from: #( 0 1 0 0.5 1.0 0.5 0 1 0 0.3 0.6 0.9 1 0.5 0.25 0.125 0.0625 0 1 0))]
```

Using Functions

One can apply a function to any property of an event list, as in the example below, which makes a crescendo/decrescendo using an exponential triangle function.

```
[ | list fcn |
list := EventList newNamed: #test3.
(0 to: 4000 by: 100) do:    "4 seconds, 10 notes per second"
      [ :index |         "add the same note"
       list add: (MusicEvent dur: 100 pitch: 36 ampl: 120) at: index].
fcn := ExponentialFunction from: #((0 0.05 2) (0.5 1 -2) (1 0.05)).
list applyFunction: fcn to: #loudness.
list inspect] d
```

The next examples show how functions can be queried; the first expression displays a function that has large changes in its slope; the expressions after that illustrate that one can ask a function for an X value where it crosses a threshold of difference from some other X value, e.g., the 2nd expression below means "find the first value after X = 0.1 where the value is more than 0.01 different from the value at X = 0.1." This is quite useful in data reduction and in output scheduling (i.e., don't keep sending the same controller value)

Create a function with a large change in slope.

```
[(LinearFunction from: #((0 0.2) (0.6 0.3) (0.62 1.0) (1.0 0.3))) edit] d
```

```
Query the delta threshold in the low-slope section
  [(LinearFunction from: #((0 0.2) (0.6 0.3) (0.62 1.0) (1.0 0.3)))
      nextXMoreThan: 0.01 from: 0.1] p
Query the delta threshold in the high-slope section
  [(LinearFunction from: #((0 0.2) (0.6 0.3) (0.62 1.0) (1.0 0.3)))
      nextXMoreThan: 0.1 from: 0.6] p
```

Functions support standard arithmetic, so that one can scale them by numbers, or add/multiply functions as though they were numbers, as illustrated by the examples below.

```
  [((FourierSummation from: #((5 0.1 0))) + 0.6) open]

  [ | sin tri |
  tri := LinearFunction from: #((0 0) (0.5 0.9) (1 0)).
  sin := FourierSummation from: #((17 0.05 0)).
  rnd := Function randomOfSize: 512 from: 0.0 to: 1.0.
  FunctionView lineWidth: 2.          "set a bold line width"
  FunctionView onFunctions:
    (Array
      with: (tri + sin)               "triangle + sine"
      with: (tri * sin + 0.2)          "triangle * sine + offset"
      with: (rnd * 0.1 + 0.5)          "noise with scale/offset"
      with: (tri * 0.25))             "triangle scaled by constant"
    withColors: (Array with: ColorValue blue with: ColorValue red
        with: ColorValue cyan with: ColorValue green)
    normalize: #(false false false false)
    x: 512 y: 256] d
```

Function Views

The following multi-linear-function view example creates 4 linear functions and displays them

```
  [FunctionView lineWidth: 2.          "set a bold line width"
  FunctionView onFunctions: (Array
      with: (LinearFunction from: #((0@0) (0.15@1.0) (0.25@0.75) (0.75@0.5) (1@0)))
      with: (LinearFunction from: #((0@0.1) (0.3@0.7) (0.7@0.45) (1@0)))
      with: (LinearFunction from: #((0@0) (0.05@1) (0.2@0.35) (1@0)))
      with: (LinearFunction from: #((0@0.2) (0.6@0.3) (1@0.3))))
    withColors: (Array with: ColorValue blue with: ColorValue red
        with: ColorValue cyan with: ColorValue green)
    normalize: #(false false false false)
    x: 512 y: 256] d
```

The next example mixes the types of function as well as the normalization switches

```
  [FunctionView
    onFunctions: (Array with: (LinearFunction from: #((0 0) (0.12 1) (0.17 0.74)
        (0.35 0.5) (0.9 0.4) (1 0) ))
      with: (FourierSummation from: #((1 0.5 0) (3 0.15 0) (5 0.1 0) (7 0.075 0) (9 0.055 0) (11 0.05 0)))
      with: (ExponentialFunction from: #((0 0 5) (0.05 1 -5) (0.2 0.25 -1) (1 0)))
      with: (SplineFunction from: #((0 0.5) (0.3 0.6) (0.7 0.5) (0.85 0.7) (1 0.6))))
      withColors: FunctionView defaultColors
    normalize: #(false true false false)
  x: 512 y: 256] d
```

Playing functions as controllers

Several of the voice classes (see below) can use Siren functions to send out sampled continuous control messages. The class FunctionEvent provides the methods needed to package functions in event lists, and allows you to specify a fixed update rate or a value change threshold for output scheduling. The following example sends out OSC frequency updates at the rate of 4 Hz to the address /osc/1/ampl with values taken from a linear envelope. Start an OSC dump utility before executing this.

```
  [OSCVoice functionExample] d
```

Similarly, one can use functions as MIDI controller messages, as in this example, which uses a linear function to create a swell on an organ note.

```
  [MIDIPort functionExample] d
```

Spectra and Signal Analysis

A spectrum is simply another kind of function (with real or complex values), and Siren supports the fast Fourier transform for spectral analysis and resynthesis via the FFTW library. There is also a simple spectral display, as illustrated by the examples below.

Create a swept sine wave and take its fft.
```
[Spectrum sweepExample display]
```

Read a file (T'ang dynasty speech) and show the spectrogram
```
[Spectrum fileExample display] db
```

Loading SHARC (Sandell Harmonic ARChive) spectral sets

Siren also has special classes that read spectral samples from Greg Sandell's SHARC timbre database (http://www.timbre.ws/sharc/files/README.txt); these sampled spectra can be used for sum-of-sines synthesis. The following example illustrates the loading of a SHARC sample for a tuba. It assumes that the SHARC database is somewhere in the user's data folder list.
```
[SHARCInstrument fromDir: 'tuba'] i
[(Function from: (((SHARCInstrument fromDir: 'tuba') samples at: #c3) asWavetable: 1024)) edit: 1024] d
[((SHARCInstrument fromDir: 'tuba') samples at: #c3) asSumOfSines] i
```

To load the entire SHARC database, use the following.
```
[SHARCInstrument loadOrchDir: 'sharc'] d
[SHARCInstrument orchestra] i
```

There's an example of using a SHARC spectrum sample set to drive a CSL sum-of-sines synthesizer via OSC in this method
```
[OSCVoice sosExample1]   db
```

Extending the Function Framework

As with other areas of Siren, the Function framework is designed for ease of extension. One can construct a new type of function with a simple instance creation method and implementation of the at: method. There are also often cases where one needs an additional behavior in the base Function class. For extending the behavior of functions with respect to events and event lists, you can customize the eventList apply: function.

Siren Event Generators

The EventGenerator and EventModifier packages provide for music description and performance using generic or composition-specific middle-level objects. Event generators are used to represent the common structures of the musical vocabulary such as chords, clusters, progressions, ostinati, or algorithms. Each event generator subclass knows how it is described--e.g., a chord with a root and an inversion, or an ostinato with an event list and repeat rate--and can perform itself once or repeatedly, acting like a Smalltalk-80 control structure or returning a static event list for further processing.

EventModifier objects generally hold onto a function and a property name; they can be told to apply their functions to the named property of an event list lazily or eagerly. Event generators and modifiers are described in more detail in the 1989 ICMC paper (see the file Doc/icmc.89.egens.pdf).

Examples

Clusters and Chords are simple one-dimensional event generators.

```
[(Cluster dur: 2.0
   pitchSet: #(48 50 52 54 56)
   ampl: 100
   voice: 1) open]

[((Chord majorTetradOn: 'f4' inversion: 0) duration: 1.0) open]
```

Rolls are also-1-D, but are rhythm-only.
Create and play a simple drum roll--another 1-D event generator.

```
[((Roll length: 2000 rhythm: 50 note: 60) ampl: 80) open]
```

Clouds are stochastic descriptions of event lists whereby one can give the numerical range of each of several standard properties.

Create and edit a low 6 second stochastic cloud with 5 events per second.

```
[ | c |
```

```
c := (Cloud dur: 6          "lasts 6 sec."
   pitch: (48 to: 60)       "with pitches in this range"
   ampl: (80 to: 120)       "and amplitudes in this range"
   voice: (1 to: 1)         "select from these voices"
   density: 5) eventList.   "play 5 notes per sec. and get the event list"
c open]
```

To create a dynamic cloud, one gives starting and ending ranges for the properties.
Play a 6-second cloud that goes from low to high and soft to loud.

```
[(DynamicCloud dur: 6
   pitch: #((30 to: 44) (60 to: 60))   "given starting and ending selection ranges"
   ampl: #((20 to: 40) (90 to: 120))
   voice: #((1) (1))
   density: 12) open]
```

A selection cloud selects values from the data arrays that are given in the instance creation method.

```
[(SelectionCloud dur: 4
   pitch: #(32 40 48 50 52 55 57 )
   ampl: #(80 40 120)
   voice: #(1)
   density: 8) open]

[(SelectionCloud dur: 4
   pitch: ((NeapolitanMinor root: N do) asPitchesInOctave: 4)
   ampl: #(80 40 120)
   voice: #(1)
   density: 12) open]
```

By obvious extension, a dynamic selection cloud allows one to specify the start and finish selection sets.
The first example blow plays a selection cloud that makes a transition from one triad to another.

```
[(DynamicSelectionCloud dur: 6
   pitch: #( #(48 50 52) #(72 74 76) )   "starting and ending pitch sets"
   ampl: #(60 80 120)
   voice: #(1)
   density: 12) open]

[ | set1 set2 |
set1 := ((Oriental root: N do) asPitchesInOctave: 4).
set2 := ((Oriental root: N sol) asPitchesInOctave: 2).
(DynamicSelectionCloud dur: 6
   pitch: (Array with: set1 with: set2)
   ampl: #(60 80 120)
   voice: #(1)
   density: 12) open]

[ | c |
c := (DynamicSelectionCloud dur: 6
   pitch: #( #(53 50 52) #(72 74 75) )   "starting and ending pitch sets"
   ampl: #(60 80 120)
   voice: #(1)
   density: 12) eventList.

c addAll: (DynamicSelectionCloud dur: 6
   pitch: #( #(76 78 80) #(60 62 64) )   "starting and ending pitch sets"
   ampl: #(60 80 120)
   voice: #(1)
   density: 16) eventList.
c open]
```

The extended DynamicSelectionCloud uses a multi-part pitch set of the format (time -> chord) (time -> chord) ... as in the following example, which creates a list of the tetrachords of the Neapolitan minor scale, then scrambles the list, then transposes every second chord up two octaves, then plays an extended dynnamic selection cloud made from this list.

```
[ | score chords list |          "generate the tetrads from the selected scale; scramble the order"
chords := ((NeapolitanMinor root: N do) generateChordsPoly: 4 inOctave: 2) scrambled.
list := ?OrderedCollection new.
```

```
1 to: 7 do:
   [ :ind |                                "shift every other one up 2 octaves"
   ind even ifTrue: [list add: ((ind - 1) * 3 -> ((chords at: ind) collect: [ :no | no + 24]))]
      ifFalse: [list add: ((ind - 1) * 3 -> (chords at: ind))]].
score := (ExtDynamicSelectionCloud dur: 8    "now make a cloud from these"
   pitch: list
   ampl: 60
   voice: nil
   density: 10) eventList.
score eventsDo: [ :ev |                      "plug in the properties for FM"
   ev inst: '/i1/pn'.
   ev modIndex: 2.0.
   ev ratio: 1.02.
   ev pos: 0.0].
SirenSession eventList: 'EvGens/dsCloud1' put: score.
score open]
```

As an example of a more sophisticated event generator, Mark Lentczner's bell peals ring the changes.

```
[(Peal upon: #(60 62 65)) eventList open]

[ | peal list |
peal := Peal upon: #(60 62 65 67).
list := EventList new.
peal playOn: list durations: 240 meter: 100 at: 0.
"   list voice: #marimba."
list open]
```

One can get an event generator's event list and process it using Smalltalk code blocks, as in this example, which takes a simple selection cloud and applies both a crescendo function and a beat pattern to it.

```
[ | dur cloud list ramp1 ramp2 pattern start |
dur := 6.
cloud := (SelectionCloud
   dur: dur
   pitch: ((NeapolitanMinor root: N do) asPitchesInOctave: 4)
   ampl: #(80)
   voice: #(1)
   density: 10).
list := cloud eventList.
SirenSession eventList: 'Examples/NeapCloud' put: list.
ramp1 := (ExponentialFunction from: #((0 0.2 5) (1 0.8))).   "2 ramps that will be summed"
ramp2 := (ExponentialFunction from: #((0 0 5) (0.2 0.2))).
pattern := #(0.1 0.1 0.2 0.2 0.1).                    "beat pattern"
start := 0.
1 to: list size do:                                "loop through the list's events"
   [ :count | | assoc |
   assoc := list events at: count.                    "get the current association"
   assoc key: start sec.                           "reset the start time"
   start := start + (pattern at: (count \\ pattern size) + 1).
   scale := (ramp1 at: (count / list size)) + (ramp2 at: ((count asFloat / list size) mod: ramp2 size)).
   assoc value ampl: (scale * 80) velocity].             "scale the amplitude"
list open]
```

Extending EventGenerators

New kinds of event generator can be programmed with constructor and eventList: methods. The framework facilitates the design of fancier behaviors and parameterized event generators.

Using EventModifiers

One can apply functions to the properties of event lists, as in the following example, which creates a drum roll and applies a crescendo modifier to it.

```
[ | roll decresc |
roll := ((Roll length: 3 rhythm: 0.06 note: 60) ampl: 120) eventList.
decresc := Swell function: (ExponentialFunction from: #((0 0 2) (1 1))).
decresc applyTo: roll.
roll open]
```

Similarly, the following changes the tempo of the drum roll.

```
[ | roll rub |
roll := ((Roll length: 5 rhythm: 0.1 note: 60) ampl: 80) eventList.
rub := Rubato function: (LinearFunction from: #((0 1) (1 0.5))).
rub applyTo: roll.
roll open]
```

or this example, where the rubato is a sine curve with an offset

```
[ | roll sin rub |
roll := ((Roll length: 5 rhythm: 0.1 note: 60) ampl: 80) eventList.
sin := (FourierSummation from: #((1 0.1 0))) + 1.0.
rub := Rubato function: sin.
rub applyTo: roll.
roll open]
```

All of these work by sending the message
```
anEventList applyFunction: aFunction to: aSelector startingAt: sTime
```
which can be used for many other situations.

There are many other examples of using functions with event lists in the MIDI, OSC, and CSL I/O examples.

Schedulers and Real-time Performance

Event lists have events sorted by their relative start times. One "performs" event lists by placing them in a schedule for performance. A schedule can have one or more client objects (usually event lists whose items are simple events, sound events, or function events) that are able to do something in response to the
```
scheduleAt: aTime
```
message. The return value from this message is assumed to be the delay (in usec) before calling the client again. Event lists typically perform the next event (by passing it to its voice) and answer the relative delta time to the next event (which may be 0 for simultaneous events).

The EventScheduler instance can be accessed by a class message (instance) to class EventScheduler; it can be used to sequence and synchronize event lists that may include a variety of events, event lists, and voices. The Scheduler messages for adding a new "client" and running the schedule are as shown below.

Play some random notes on the devault voice
```
[EventScheduler addClient: (EventList randomExample: 20) at: 500 msec.
 EventScheduler run]
```

Run the simple flashing rectangles example
```
[EventScheduler addClient: ActionEvent listExample in: 100 msec; run]
"Window currentWindow refresh"
```

House-keeping messages
```
[EventScheduler isRunning]
[EventScheduler interrupt]
[EventScheduler flush]
[EventScheduler initialize]
```

Get the instance
```
[EventScheduler instance]
```

If the SirenUtility vertbosity is set to 2 (very verbose), and you hold down the key, the schedule will print a timer to the Transcript (useful for testing).

The standard voices for MIDI and OSC output use the built-in schedule for their timing.
Play 64 notes lasting 80 msec--a good test of real-time performance.
This is scheduled in the port, i.e., at the lowest-possible level.
```
[MIDIPort testRandomPlay2: 64 dur: 80]
```

Test a roll--it's easier to hear scheduler jitter here. The first example uses the low-level port delays. (try it several times to hear the changes in the jitter.)
```
[MIDIPort testRoll: 40 dur: 60]
```

This should sound about the same, but plays an event generator over the the high-level scheduler.
```
[((Roll length: 2400 rhythm: 60 note: 60) ampl: 96) play]
```

This example uses the high-level EventScheduler to play a scale. (Jitter is harder to hear here.)

```
[MIDIDevice scheduleExample]
```

Scheduler Internals

The EventScheduler instance (Schedule) holds onto 2 SEventQueues, one for normal clients, and one for timers. The SEventQueue is a simple doubly-linked list of ScheduleRecord objects sorted by start time (with a class pool of ScheduleRecords and an efficient insert method). All times are in microseconds (meaning long integers).

The main scheduler loop is in the method run forks a process running the loop

```
[running] whileTrue: [self callNextAppointment]
```

It's the callNextAppointment method that looks for a client or timer that's ready and schedules it. If there's nothing to do, the loop sleeps a bit; the actual amount of the inner delay is a class variable.

Voices and Ports in Siren

The "performance" of events takes place via voice objects. Event properties are assumed to be independent of the parameters of any synthesis instrument or algorithm. A voice object is a "property-to-parameter mapper" that knows about one or more output or input formats for Smoke data. There are voice "device drivers" for common file storage formats--such as note lists file formats for various software sound synthesis packages or MIDI files--or for use with real-time schedulers connected to MIDI, OSC, CSL, or SuperCollider drivers. These classes can be refined to add new event and signal file formats or multilevel mapping (e.g., for MIDI system exclusive messages) in an abstract way.

Voice objects can also read input streams (e.g., real-time controller data or output from a coprocess), and send messages to other voices, schedulers, event modifiers or event generators. This is how one uses the system for real-time control of complex structures.

The actual property-to-parameter mapping is often controlled by a dictionary or a block (the parameter map) that takes the properties of an event and creates a statement or command for some output format. This allows the user to customize the voices at run-time (see the OSCVoice for good examples).

Voices and Schedulers

Some voices are "timeless" (e.g., MIDI file readers); they operate at full speed regardless of the relative time of the event list they read or write. Others assume that some scheduler hands events to their voices in real time during performance. The EventScheduler does just this; it can be used to sequence and synchronize event lists that may include a variety of voices.

Examples

Create a random event list and write it out to notelist files in any of several formats. Edit the file.

```
[CmixVoice randomExampleToFileAndEdit]
[CmusicVoice randomExampleToFileAndEdit]
[CsoundVoice randomExampleToFileAndEdit]
[SuperColliderVoice randomExampleToFileAndEdit]
```

Create an event list of 20 notes with semi-random values and play it on a MIDI output voice.

```
[(EventList randomExample: 20) playOn: MIDIVoice default]
```

Use the same random list creation method, but add three lists in parallel.

```
[((EventList newNamed: #pRand)
      addAll: (EventList randomExample: 40);
      addAll: (EventList randomExample: 40);
      addAll: (EventList randomExample: 40))
   playOn: MIDIVoice new]
```

Complex Multimedia Example: Generate and play a mixed-voice event list; a cloud plays alternating notes on MIDI and built-in synthesis, and a list of action events flashes screen rectangles in parallel.

```
[ | el |
el := (Cloud dur: 6            "Create a 6-second stochastic cloud"
   pitch: (48 to: 60)           "choose pitches in this range"
   ampl: (40 to: 70)            "choose amplitudes in this range"
                    "select from these 2 voices"
   voice: (Array with: (MIDIVoice default) with: (OSCVoice default))
   density: 5) eventList.        "play 5 notes per sec. and get the events"
                  "add some animation events"
el addAll: ActionEvent listExample.
```

```
el play]                    "and play the merged event list"
```

Defaults

The Voice class has a default subclass. This can be changed with the SirenUtility GUI.

```
[Voice default] i
```

This can be changed with the Siren Utility view.
As with EventLists, Voices can also be stored in a global dictionary and acccessed by name.

```
[SirenSession voice: #oscFM put: (OSCVoice map: #pMapForCSLSimpleFM)] d
[SirenSession voice: #defaultMIDI put: (MIDIVoice named: 'oboe' onDevice: (MIDIDevice on: 1) channel: 1)] d
[SirenSession voiceNamed: #oscFM] i
```

MIDI Input Voices

The MIDI file voice can read standard MIDI file format and generate a Siren event list, as in the example below.

```
(MIDIFileVoice newOn: 'K194.MID')
   readOnto: (EventList newNamed: #K194).
(EventList named: #K194) open.
```

Extending the Voice Framework

By adding new voice classes, you can add parsers and generators for new score file formats, or extend the system with new real-time output driver interfaces. Look at the uses of the parameter maps in the notelist voices, or the OSC and CSL voices.

About Siren MIDI

Siren includes a portable MIDI I/O framework that consists of an abstract I/O port class (MIDIPort), a plug-in that uses the DLLCC interface, and a C-language interface module that talks to the platform-independent PortMIDI library. The higher-level model is that a MIDI voice object holds onto a MIDI device and a channel. The MIDI device object is connected to a MIDI port. For example, the verbose way to create the default MIDI voice would be to say

```
MIDIVoice on: (MIDIDevice on: (MIDIPort default openOutput))
```

The voice object gives us the standard voice behavior (like event mapping and scheduling). The MIDI device allows us to model the device-specific messages supported by some devices. (I used to use micro-tonal extended messages on a few different hardware synths.) The MIDIPort is used for the interface betweeen Siren and external MIDI drivers and devices. It implements both note-oriented (e.g., play: pitch at: aDelay dur: aDur amp: anAmp voice: voice), and data-oriented (e.g., put: data at: delay length: size) behaviors for MIDI I/O.

There is typically only one instance of MIDIPort; the messages new, default, and instance all answer the sole instance. MIDIPorts use observers (dependency) to signal input data--objects wishing to receive input should register themselves as dependents of a port. In the default Siren implementation, the scheduler is all in Smalltalk, and only the simplest MIDI driver is assumed.

MIDI Implementation: The class PortMIDIPort implements the low-level MIDI I/O messages by talking to the PortMidiInterface external class, which is a front-end to C-language glue code that talks to the PortMIDI library. If you set the verbosity to 2 and open a port, it will print your entire device table to the VM's standard output and to the Transcript; for my system, this looks like the following:

```
Midi Device Table
   0: IAC Driver IAC Bus 1 - in
   1: IAC Driver IAC Bus 2 - in
   2: IAC Driver IAC Bus 3 - in
   3: PC-1600X - in
   4: StudioLogic SL-161 - in
   5: Tascam FW-1804 FW-1804 Control Port - in
   6: IAC Driver IAC Bus 1 - out
   7: IAC Driver IAC Bus 2 - out
   8: IAC Driver IAC Bus 3 - out
   9: Tascam FW-1804 FW-1804 Port 1 - out
   10: EM-100 - out
   11: Tascam FW-1804 FW-1804 Port 3 - out
   12: Tascam FW-1804 FW-1804 Port 4 - out
   13: Tascam FW-1804 FW-1804 Control Port - out
```

MIDI Tests and Examples

Basic Tests

Edit the methods MIDIPort initialize to suit your setup.

Try to open and close the MIDI port (this also reports to the transcript and dumps a device list to the VM's stdout).
[MIDIPort testOpenClose]

Open MIDI, play a 1-sec. note.
[MIDIPort testANote]

Open MIDI, play a fast scale.
[MIDIPort testAScale]

Open MIDI, play notes based on the mouse position (x --> voice; y --> pitch) until mouse down.
[MIDIPort testMouseMIDI]
[MIDIPort allNotesOff]

Close down and clean up.
[MIDIPort cleanUp]

Using voices
[MIDIVoice randomExample]
[MIDIVoice scaleExample]
[MIDIVoice voiceInspect]

General MIDI Maps and Program Changes

"Demonstrate program change by setting up an organ instrument to play on.
[MIDIPort testProgramChange]

Down-load a general MIDI patch for a 4-voice organ.
[MIDIPort setupOrgan. Cluster example1]

Down-load a general MIDI patch for a 16-voice percussion ensemble.
[MIDIPort setupTunedPercussion. MIDIPort testAScale]

Or try these
[MIDIPort setAllInstrumentsTo: 'Tenor Sax'. MIDIPort testAScale]
[MIDIPort setAllInstrumentsTo: 'Music Box'. MIDIPort testAScale]

Reset the GM map (for the first 16 instruments)
[MIDIPort setupDefaultGeneralMIDI]

MIDI Input

Open MIDI, try to read something--dump it to the transcript.
[MIDIPort testInput]
[MIDIPort dumpExample]

Execute this to end the input test
[MIDIPort testInputStop]

Get the port's pending input.
[MIDIPort default eventsAvailable]
[MIDIPort default readAll]
[MIDIPort default input]
[MIDIPort default resetInput]

Set up a MIDI dump object as a dependent of the input port. Dump for 5 seconds,
then turn off. The default update: method just dumps the MIDI packet into the transcript.
[MIDIPort dumpExample]

This example captures notes to an event list for 5 seconds and opens an editor on it.
[MIDIDump exampleList]

Set up uncached controller reading and dump input to the transcript.
[MIDIPort testControllerInput]
[MIDIPort testInputStop]

Set up uncached controller reading--read controllers from lo to hi as an array and print it;
stop on mouse press.

```
[MIDIPort testControllerCachingFrom: 48 to: 52]
```

Real-time Performance Tests

Play "num" random pitches spaced "dur" msec apart.
This test creates the messages and does the scheduling right here.

```
[MIDIPort testRandomPlayLowLevel: 64 dur: 80]
```

Play a roll of 'num' notes spaced 'dur' msec apart.
This test creates the messages and does the scheduling right here.

```
[ObjectMemory compactingGC.
   MIDIPort testRollLowLevel: 20 dur: 80]

[ObjectMemory compactingGC.
   MIDIPort testRollLowLevel: 200 dur: 40]
```

Continuous Control Tests

Demonstrate control commands by playing a note and making a crescendo with the volume pedal.

```
[MIDIPort testControlContinuous]
```

Demonstrate pitch-bend by playing two notes and bending them.

```
[MIDIPort testBend]
```

Recording Continuous Controllers

One can also record functions from input controlers, as in the following example, which reads MIDI controller 48 at a rate of 40 Hz for 5 seconds.

```
[MIDIPort testControllerRecording]
```

Utilities

ANO

```
[MIDIPort allNotesOff]
```

Close down and clean up

```
[MIDIPort cleanUp]
```

If things get wedged, do this

```
[PortMidiInterface unloadLibraries]
```

For much more detail, see the class example messages in MIDIPort, or the tests in the PortMIDIPort and PortMidiInterface classes.

Siren and OpenSoundControl

Siren includes an output voice that generates messages in the CNMAT OpenSoundControl (OSC) protocol (http://www.cnmat.berkeley.edu/OpenSoundControl), which is sent out via UDP network packets to some synthesis server. We generally build these servers using CSL or SuperCollider, and then control them with set-up, event trigger, and control messages sent out from Siren.

The verbosity flag in the SirenUtility class allows for logging of all OSC to the transcript; open a Siren utility panel and use the pop-up menu to set the verbosity to 2. To test the OSC I/O, look at the classes OSCPort and OSCVoice with the default host/port settings and several useful parameter maps. A parameter map is a Smalltalk block that takes an event as its argument and returns an OSC message object (or a bundle), as in the following example (don't execute this):

```
[ :event | | arr |          "This block takes an event as its argument and answers a"
arr := Array              "TypedOSCMessage for the address /note-on"
      with: event duration asSec value
      with: event pitch asHz value
      with: event ampl asRatio value.
TypedOSCMessage for: '/note-on' with: arr]
```

For simple debugging, Chandrasekhar Ramakrishnan wrote Occam, a stand-alone OSC-to-MIDI convertor for Mac OS X (http://www.mat.ucsb.edu/~c.ramakr/illposed/occam.html). You can also use the CNMAT dumpOSC utility to print out OSC messages. The following example demonstrates using OSC with the Occam convertor, to test OSC output using a MIDI synthesizer.

```
[OSCVoice midiScaleExample]
```

There are also several examples that are set up to work with the CSL OSC server demos; if you have CSL, compile and start the "OSC_test" target, which sets of a simple server with 4 voices of FM synthesis and 4 sound file playback instruments. Look at the conditional compilation macros in the file OSC_main.cpp; there are several options, each of which compiles a different instrument library into the CSL OSC server.

```
#define CSL_OSC_FM_SndFile      // 4 voices of FM, 4 of SndFiles, and 1 bell
//#define CSL_OSC_SAMPLER       // 16 voices of file playback
//#define CSL_OSC_ADDER         // 16 voices of sum-of-sines synthesis
```

Then you can try the OSCVoice examples that follow.

```
[OSCVoice fmExample1]
[OSCVoice sndExample1]
```

These examples loop endlessly, so you have to interrupt or flush the scheduler to stop them

```
[OSCVoice fmExample2.       5 wait.
OSCVoice sndExample2.       5 wait.
OSCVoice fmExample4]
 [EventScheduler flush]
```

Siren OSC also supports control output, as in this example, which sends values from a linear envelope out to the address "/osc/1/ampl" at the rate of 4 Hz (use dumpOSC to view the results).

```
[OSCVoice functionExample]
```

You could rewrite this to use the function's change threshold instead of a constant update rate.

As an example that mixes both styles, the following expression plays a long low FM note and then uses Siren function objects to send continuous controls to make the note glissando down and pan from left to right.

```
[OSCVoice fmExample3]
```

For more examples, set the default voice to OSC (using the Siren utility control panel) and run the built in examples on the other pages of this outline.

Sound Objects in Siren

Siren's hierarchy of events and functions includes objects that represent sampled sounds. These can be used for synthesis, recording, processing, and playback. The fact that a Siren sound is a function means that it has the semantics of a single-valued function of time. A concrete SampledSound has something in its data instance variable (inherited from Function) that might be a word (16-bit) or floating-point sample array, or a CPointer, meaning that the sound's actual data is held onto by CSL, PortAudio, LibSndFile, or Loris. Since sounds are also events, they can have properties such as sound metadata or control functions.

There are many instance creation methods in the class SampledSound, including examples to create several kinds of waveforms, frequency sweeps, and impulse trains.

Examples

Create a 1-second sine wave sound at a sample rate of 44100 Hz, with 1 channel and the base frequency of 80 Hz.

```
[(SampledSound sineDur: 5 rate: 44100 freq: 80 chans: 1) edit]
```

View a swept sine wave

```
[(SampledSound sweepDur: 2.0 rate: 44100 from: 10 to: 100 chans: 1) edit]
```

View a pulse train

```
[(SampledSound pulseTrainDur: 5.0 rate: 44100 freq: 200 width: 0.1 chans: 1) edit]
```

View a sawtooth waveform (these 2 methods are the same)

```
[SoundView openOn: SampledSound sawtooth]
[SampledSound sawtooth edit]
```

Read in a sound from a file

```
[(SampledSound fromFile: 'unbelichtet.aiff') edit]
```

Save a sound to a file

```
[(SampledSound sweepDur: 5.0 rate: 44100 from: 50 to: 1000 chans: 1)
```

```
        scaleBy: 0.2; storeOnFileNamed: 'sweep.aiff']
  [(SampledSound fromFile: 'sweep.aiff') edit]
```

Load and edit a long-ish sound file

```
  [(SampledSound fromFile: 'FourMagicSentences.aiff') edit]
```

Manipulating Sound Objects

Sampled sound objects can be traeated as normal functions, i.e., one can address them as sample arrays and perform all manner of sample arithmetic, as illustrated in the following example, which mixes a sine wave and a sawtooth using low-level sample-accessing messages.

```
  [ | sin saw |
  sin := SampledSound sineDur: 1.0 rate: 44100 freq: 10 chans: 1.
  saw := SampledSound sawtoothDur: 1.0 rate: 44100 freq: 100 chans: 1.
   sin scaleBy: 0.8.
   saw scaleBy: 0.1.
   1 to: sin size do:          "loop to do verctor math on sound samples"
     [ :index |
      sin at: index put: ((sin at: index) + (saw at: index))].
   sin edit]
```

Class SampledSound also supports basic envelope extraction, as in this example,

```
  [(SampledSound fromFile: 'kombination1a.snd') rmsEnvelope edit]
```

As a final example, one can apply a function to a sound as an envelope, as in this block,

```
  [((SampledSound sineDur: 1.0 rate: 44100 freq: 220 chans: 1)
        scaledByFunction: (ExponentialFunction default)) edit]
```

For fancier signal processing, we use the CSL package rather than sample arithmetic with sounds and fnuctions.

FFTs and Spectra

There's also an FFT-based spectrum class in Siren, which uses the external interface to the FFTW FFT package. To use it, look at the example methods in the class Spectrum and SpectrumView

```
  [Spectrum sweepExample display]
  [Spectrum fileExample display]
```

For more on sound processing, see the the various *sound* class utility methods, the SoundFile I/O methods, and notes below on the CSL framework in Siren.

Sound File I/O

Siren sound objects can be read from and written to sound files using an external interface to Eric DeCastro's libSndFile library. This supports all popular (and many very obscure) sound file formats.

Store a swept sine to a file

```
  [(SampledSound sweepDur: 2.0 rate: 44100 from: 30 to: 300 chans: 1)
       storeOnFileNamed: 'sweep.aiff']
```

Look at the file using your favorite sound file editor, or do

```
  [(SampledSound fromFile: 'sweep.aiff') edit]
```

Read various file formats

```
  [(SampledSound fromFile: 'unbelichtet.aiff') edit]
  [(SampledSound fromFile: 'kombination1a.snd') edit]
```

Streaming Sound Record/Playback

To support real-time streaming sound recording and playback, an external interface is provided to the cross-platform PortAudio library. The singleton instance of PortAudioPort communicates with the external driver. As with the MIDI port, the sound port object loads and maintains a device table. For my system, this is,

```
Sound Device Table
  1: Built-in Microphone 2 in 0 out 44100 Hz
  2: Built-in Line Input 2 in 0 out 44100 Hz
  3: Built-in Output 0 in 2 out 44100 Hz
```

4: TASCAM FW-1804 8 in 8 out 96000 Hz

To play a sound file using the default external utility , use
[Sound playFile: 'unbelichtet.aiff']

(see the method SirenUtility playSoundFile: to change the actual command line,)

This example plays a 3-second sine wave sweep
[PortAudioPort playSweep]

Same only 20 seconds long
[PortAudioPort playSweepLong]

The class SoundPort is abstract and has two subclasses: PortAudioPort and SmartAudioPort. SmartAudioPort uses call-backs from the C callback into Smalltalk. It works for recording and playback, but is still buggy (clicks a lot); see the class examples.

The Siren Graphics Framework

The Siren graphical applications are based on the simple display list graphics framework in the categories MusicUI-DisplayLists and MusicUI-DisplayListViews. This package includes display items such as lines, polygons, curves, text items, and images, hierarchical display lists, and display list views, editors, and controllers. The display list view/controller/editor are MVC components for viewing and manipulating display lists. Simple examples of the display list framework are given below.

There are several layouts for the zoom/scroll bars; in the default layout, the bars are grouped on the left and bottom of the window. The zoom bars are gray sliders on the outside, and the scroll bars are the usual color and look, and are set inside of the zoom bars. Take a look at the following and use the zoom/scroll bars.
[DisplayList rectangleExample]

Note that the small button labeled "z" in the upper-left of the window zooms back to 1@1 scale.

An alternative layout (which I prefer) places the zoom bars on the top and right. look at,
[DisplayListView open4SquareOn: (DisplayList rectanglesX: 4000 byY: 4000)]
[DisplayListView open4SquareOn: DisplayList randomExample]

The pop-up menu has many functions that are not implemented in the top-level display list view.

Display random strings
[DisplayList stringExample]

Display a hierarchical list
[DisplayListView exampleHierarchical]

[DisplayList randomExample display]

There are many more examples in the display item classes, and the display list view hierarchy.

Layout Managers and Navigator MVC

The Siren version of "Navigator MVC" framework is based on layout manager objects that can generate display lists from structured objects. This enables, for example, a variety of musical notations.

LayoutManagers take data structures and generate display lists based on their layout policies. For example, to see a class inheritance hierarchy as an indented list, use an IndentedListLayoutManager as in,

[DisplayListView colorClassListExample]

(Note that color denotes species in this example.)

To view the same structure as a tree-like layout, use an IndentedTreeLayoutManager, as in,

[DisplayListView classTreeExample]

Graphical Forms

Siren includes a hierarchical dictionary of images for use in musical notations. Execute the following to display the various forms. The method below steps through the form dictionaries and displays them in a window.

[DisplayVisual displayMusicConstants]

Siren GUI Applications

Quick tour: http://fastlabinc.com/Siren/Doc/Siren.GUI_2007.html

Based on the display list view framework and the Navigator layout managers, Siren implements a variety of musical notations. Layout managers serve as the basis for Siren's music notation applications. The basic event-oriented layout manager uses the horizontal axis to denote time (flowing from left to right), as in the next example, which opens a time sequence view on a random event list.

```
[TimeSequenceView randomExample]
```

In the time sequence view, the "note head" signifies the event's voice, not the duration. Try zooming this view.

A pitch/time view is an extension of this that uses the vertical dimension to display an event's pitch, as in piano-roll notation; for example, to display a pitch/time view on a 3-stream event list, try,

```
[PitchTimeView randomExample]
[PitchTimeView openOnEventList: (EventList scaleExampleFrom: 48 to: 84 in: 10)]
```

In the above example, the note heads denote the events' voices, horizontal blue lines originating at the note heads show the events' lengths, and red vertical lines show the events' amplitudes. To see how this is done, look at class PitchTimeView's various implementation of the itemFor: method.

Open a pitch/time view on a *very long* 3-stream event list.

```
[PitchTimeView randomExampleLong]
```

A more complete example is Hauer-Steffens notation, which has a clef and staff lines as in common-practise notation.

```
[HauerSteffensView randomExample]
[HauerSteffensView randomSWSSExample]
[(EventList scaleExampleFrom: 48 to: 60 in: 3) edit]
```

Test panning and zooming these examples.

Sound View

The sound view is a simple waveform display. One can scroll, zoom, and edit.
Use the pop-up menu to create sound objects based on a number of standard synthesis methods.

```
[(SampledSound sweepDur: 3.0 rate: 44100 from: 10 to: 400 chans: 1) edit]
[SoundView openOn: (SampledSound fromFile: 'kombination.snd')]
```

File Browser Extensions

Select a sound or score file in the file browser and note the new tab in the file pane. this allows you to play, edit, or remove files easily.

See also

FunctionViews
Loris GUI Examples
LPC GUI Examples

Extending Siren's MVC Framework

One can of course easily build new notations and new editors with the Siren MVC framework and GUI widget set. New layout managers generally consist of less than 1 page of code, and new controller/editor functions are quite simple to add. Most new tools consist of composite panes that use the simple Siren display widgets as components, and add fancier editor behaviors.

Using the Siren Utility Pane and Transports

Support classes SirenUtility and SirenSession

There are two classes with utility and session management methods: SirenUtility and SirenSession.

SirenUtility is never instantiated, but has class methods for a number of usefiul utility functions related to system configuration (global verbosity flag and search directories). There are special functions for file search (see the section

below on this page) and for handling s7 files (see below as well). The SirenUtility class initialization method sets the global logging verbosity level and creates a couple of lists of directories for searching for sound or score files. Users will typically customize this method so that Siren can find their data files.

SirenSession holds onto the user's persistent data (e.g., sounds and scores) in class-side dictionaries. This means that, by default, every sound or score that you give a name to becomes persistent and is stored in your Smalltalk virtual image. There are special functions for loading these caches from your hard disk, or for flushing them.

Links to the EventScheduler

The SirenSession also registers itself as a dependent of the EventScheduler, so it handles event logging and transport functions.

SirenUtility Panel

There are two main GUIs for Siren configuration: the Utility and the Transport.

The Siren utility panel has buttons for configuring and testing MIDI, sound, and OSC I/O, as well as for setting some global values such as the logging verbosity. There are also buttons in this panel for loading and flushing the session data.

To open the Siren utility view, use the button in the Launcher, or execute
```
[SirenSession openUtility] d
```

The left-most buttons in the pane asre for setting up and testing the MIDI and OSC I/O defaults. The 2nd row is for sound I/O cnofiguration.

SirenTransport Panel

The other SirenSession class GUI is the transport panel; to open this, hold down while pressing the Siren utility button in the launcher, or execute,
```
[SirenSession openTransport] d
```

The transport has menu buttons along the left for accessing your data (sounds, scores, and voices) and for controlling the EventScheduler. Below are some useful utility messages.

```
"SirenSession release"
"SirenSession instanceCount"
"SirenSession allInstances do: [ :ss | ss release]"
"EventScheduler instanceCount"
"EventScheduler instance"
```

SirenUtility file tricks

The folliowing examples demonstrate the SirenUtility class support for sound/score data. As mentioned above, users can customize the search directories by editing the SirenUtility class initialize method.

Find a file with the given name in any of the user's sound/score folders

```
[SirenUtility findFile: 'stp.ev'] p
```

Find a directory

```
[SirenUtility findDir: 'Data' tryHard: false] p
[SirenUtility findDir: 'Frameworks' tryHard: true] p
```

List all files with the given extension

```
[SirenUtility findFiles: 'au'] p
[SirenUtility findFiles: 'mid'] p
```

Create a new s7 folder and ask for the next free of a specific type name in it

```
[SirenUtility createS7: ((SirenUtility findDir: 'Data/'), 'testing')] d
[SirenUtility nextName: 'testing' type: 'aiff' ] p
```

SirenSession Storage
```
SirenSession eventLists
SirenSession sounds
```

Recreate the instance
```
SirenSession refresh
```

SirenSession release
SirenSession instance

Siren DLLCC External Interfaces

Siren includes several interfaces between Smalltalk and C to access sound and MIDI I/O in a cross-platform manner. These use the DLLCC package to generate Smalltalk classes whose methods are references to C functions in a dynamic library.

To compile and load these, you should either down-load the CSL library binaries package from the CSL home page (http://fastlabinc.com/CSL) or else you need to be able to use your platform's C compiler and linker (a Makefile is provided), and to capture the standard output from a VisualWorks virtual machine (normally routed to a console window).

Load and test the DLLCC External Interfaces
LibSndFile - sound file IO in many formats
[LibSndFileInterface example1: 'unbelichtet.aiff']
PortMIDI - cross-platform MIDI API
[PortMidiInterface testMIDI]
PortAudio - cross-platform audio API
[PortAudioInterface example0]
FFTW - Fast Fourier Transform
[FFTWInterface example] (try this twice)

Look at each of the interface class defs in MusicIO-External and plug in the directory names for your dynamic libraries. Each of these tests dumps output to the transcript and to the VM's stdout/console.

Flush all
[SirenExternalInterface unload]

LibSndFile - see the class def for Siren.LibSndFileInterface in category MusicIO-External; tune the lines

```
#(#libraryFiles #('libsndfile.dylib' 'sndfile_lite.dylib'))
#(#libraryDirectories #('Siren7.5/DLLCC' '/usr/local/lib'))
```

This test dumps the first few 100 samples of the test file to the transcript.
[LibSndFileInterface example1: 'unbelichtet.aiff']
"LibSndFileInterface unloadLibraries"

PortMIDI - see the class def for Siren.LibSndFileInterface in category MusicIO-External; tune the lines

```
#(#libraryFiles #('portmidi_lite.dylib' 'portmidi.dylib' 'CoreMIDI'))
#(#libraryDirectories #('/usr/local/lib' 'Siren7.5/DLLCC'
        '/System/Library/Frameworks/CoreMIDI.framework/Versions/Current'))
```

This test prints the return values of a few simple MIDI operations to the transcript.
Look in the standard output of the VM (on the Mac, this means open a console tool) to see the device list that is dumped by the initialize method.
[MIDIPort testOpenClose]
[PortMidiInterface testMIDI]
[MIDIPort testANote]
"PortMidiInterface unloadLibraries"

PortAudio - see the class def for Siren.PortAudioInterface in category MusicIO-External; tune the lines

```
#(#libraryFiles #('portaudio_lite.dylib' 'libportaudio.dylib'))
#(#libraryDirectories #('Siren7.5/DLLCC' '/usr/local/lib'))
```

These tests are progressive, first open/close only, then simple call-backs, then actual data transfer
Look in the standard output of the VM (on the Mac, this means open a console tool) to see the device list that is dumped by the initialize method.
[PortAudioInterface example0]
[PortAudioInterface example1]
[PortAudioPort playSweep]
"PortAudioInterface unloadLibraries"

FFTW - see the class def for Siren.FFTWInterface in category MusicIO-External; tune the lines

```
#(#libraryFiles #('fftw_lite.dylib' 'libfftw3f.a'))
#(#libraryDirectories #('/usr/local/lib'))
```

The basic test creates a sawtooth wave and takes its FFT, displaying the result in a function view. The view also displays the original signal if you hold down while executing the expression.

```
[FFTWInterface example]
[Spectrum sweepExample display]
[Spectrum fileExample display]
```

There are more examples of the use of FFTW in the Spectrum class.

See also the sections below on the SWIG interfaces to CSL and Loris.

Loading the SWIG Packages in Siren

There is a Smalltalk back-end to the SWIG interface generator ported by Ian Upright. This allows us to use the SWIG APIs to access CSL and Loris packages from within Siren. If you're new to this, look at the README files in the SWIG example folders, then at the example notes below. See esp.
http://commonsmalltalk.wikispaces.com/SWIG+Documentation
and
http://commonsmalltalk.wikispaces.com/SwigForVW

Note that normal Siren uesrs don't need to go into this; you can use the CSL and Loris interfaces (see the workbook section on models below) without having to regenerate the SWIG interfaces.

Testing the Smalltalk SWIG Interface using Ian Upright's Example code

Basic SWIG test: shared constants in the API - First you have to go to the example folder (Siren7.5/SWIG_Smalltalk/Examples/smalltalk/constants) and compile the example library, producing a .dll or .a file. The code block below calls the AMLImporter to load a list of .ssi files, creating an external interface and group of classes in the Siren namespace. (Edit your folder name in below.)

```
[#( 'ExampleConstants' 'ExampleConstantsConstants' 'ExampleConstantsEmbeddedConstants'
    'ExampleConstantsNI' 'ExampleConstantsTest' )
     do: [:nam |
       AMLImporter new
            baseDirectoryName: 'Siren7.5/SWIG_Smalltalk/Examples/smalltalk/constants/' ;
           namespace: Smalltalk.Siren ;
           import: nam]]
```

Look at the class def for ExampleConstantsNIExternalInterface and make certain the DLL name and folder are correct; I have to paste in,

```
        #(#libraryFiles #('ExampleConstants.a'))
         #(#libraryDirectories #('Siren7.5/SWIG_Smalltalk/Examples/smalltalk/constants'))
```

Now test is with (this prints stuff to the transcript)

```
[Siren.ExampleConstantsTest run] d
```

Next step: variables in C. Compile the ExampleVariables library as above, then execute the following block.

```
["#( 'ExampleVariables' 'ExampleVariablesNI' 'ExampleVariablesTest' 'SwigTypePint' 'SwigTypePPoint' )"
#( 'ExampleVariablesTest' ) do: [:nam |
AMLImporter new
   baseDirectoryName: 'Siren7.5/SWIG_Smalltalk/Examples/smalltalk/variables/' ;
   namespace: Smalltalk.Siren ;
   import: nam]]
```

Look at the class def for ExampleVariablesNIExternalInterface, editing it to fit your environment; then try,

```
[Siren.ExampleVariablesTest run] d       "prints stuff to the transcript"
```

Now try loading a C++ class hierarchy and creating some shape objects

```
[ #( 'ExampleClassTest' ) do: [:nam |
AMLImporter new
   baseDirectoryName: 'Siren7.5/SWIG_Smalltalk/Examples/smalltalk/class/' ;
   namespace: Smalltalk.Siren ;
   import: nam]]
```

Look at the class def for ExampleClassNIExternalInterface

```
Siren.ExampleClassTest run       "prints stuff to the transcript"
```

Using Loris and CSL via SWIG

Loris is a sound analysis/resynthesis package written by Kelly Fitz that uses the model of time-reassigned bandwidth-enhanced partial lists to allow one flexible resynthesis control and sound morphing. Siren uses the SWIG-generated interface and glue code to create Smalltalk classes with methods for modeling Loris analyzers and partial lists. To get the current Loris release, see http://sourceforge.net/projects/loris

To build it from scratch (not necessary unless you change the C++ code), first declare the shared variable for the interface, then load all the SSI files.

```
[Siren.Loris defineSharedVariable: #LorisNI
   private: false
   constant: false
   category: 'external'
   initializer: nil
   attributes: #( #(#package 'Siren'))] d
```

To import the Loris interface (the SSI files generated by swig), execute the following.

```
[AMLImporter new
   baseDirectoryName: 'Siren7.5/SWIG_Loris/';      "Edit your folder name in here"
   namespace: Smalltalk.Siren.Loris ;              "put it in its own namespace"
   import: 'Loris'] d
```

Look at the class definition for LorisNIExternalInterface and make certain the DLL or dylib name is right for your platform, and that the path is correct; I paste in the following lines,

```
        #(#libraryFiles #('loris_smalltalk.dylib'))            "Edit your DLL name in here"
        #(#libraryDirectories #('Siren7.5/SWIG_Loris'))       "Edit your folder name in here"
        #(#beVirtual false)
        #(#optimizationLevel #full))
```

and accept the change. You might want to file in ./Siren7.5/Loris-freeModule.st

For normal usage, set-up with this

```
   [Siren.Loris.Loris initializeModule] d
   [Siren.Loris.LorisNI] i          "inspect the interface if you're interested"
```

Test with this (prints the Loris version string)

```
   [Siren.Loris.Loris version] p
```

Clean up utilities

```
   Siren.Loris.Loris freeModule
   Siren.Loris.LorisNIExternalInterface unloadLibraries
   Siren.Loris.LorisNIExternalInterface instanceCount
   Siren.LorisSound instanceCount
   Siren.LorisSound allInstances
```

Using the SWIG Interface to CSL

The CREATE Signal Library (CSL) is a C++ framework for building digital audio synthesis and processing applications. Siren uses the SWIG-generated CSL interface and glue code together with CSL.dylib to create CSL object networks (DSP flow graphs) from within Smalltalk.

Get CSL

```
   http://FASTLabInc.com/CSL
```

The C++ code still has lots of multiple inheritance (i.e., Scalable, Phased, etc.), which makes some problems in the generated (single inheritance) Smalltalk models; I've added some utility messages to UnitGenerator (e.g., setScale: and setOffset:), but these might cause problems if used incorrectly.

To build it from scratch (not necessary unless you change the C++ code), first declare the shared variable for the interface, then load all the SSI files.

```
[Siren.CSL defineSharedVariable: #CslNI
   private: false
   constant: false
   category: 'external'
   initializer: nil
   attributes: #( #(#package 'Siren'))] d
```

```
[AMLImporter new
   baseDirectoryName: 'Siren7.5/SWIG_CSL/';              "Edit your folder name in here"
   namespace: Smalltalk.Siren.CSL ;
   import: 'CSL']
```

Look at the class definition for LorisNIExternalInterface and make certain the DLL or dylib name is right for your platform, and that the path is correct.

```
        #(#libraryFiles #('CSL_SWIG_wrap.so'))
        #(#libraryDirectories #('Siren7.5/SWIG_CSL'))      "Edit your folder name in here"
        #(#beVirtual false)
        #(#optimizationLevel #full))
```

To test CSL, try the following,

Set-up

```
   [Siren.CSL.CSL initializeModule] d
```

First test; print a random number

```
   [Siren.CSL.CSL fRand    ] p
```

Test the logging

```
   [Siren.CSL.CSL logMsg: 'Testing logging from Smalltalk'] d
```

House-keeping

```
   [Siren.CSL.CslNI] i
   Siren.CSL.CSL freeModule
    Siren.CSL.CslNIExternalInterface unloadLibraries
    Siren.CSL.CslNIExternalInterface instanceCount
```

The CSL IO classes cache the singleton IO object, so you might have to explicitly flush it from time to time

```
   [Siren.CSL.IO release]
```

Object Models for Loris and CSL

The Siren kernel of the Loris interface is in the classes LirosSound and LirosAnalyzer. The class LorisAnalysisConfiguration represents the variables used by the analyzer.

Simple Loris tests

To load a sound file and run it, saving the result as SDIF and opening a display (read this method to see how to use the Loris API)

```
   [LorisSound example: '1.2a1.aiff' resolution: 70] db
   [LorisSound fromSDIF: 'horns1c.sdif']
```

Loris Analyzer Configuration

There is a special class that manages the Loris analyzer settings; it has an editor default Loris analyzer settings (when set to resolution of 70 Hz) are:

```
   hopT 0.0142857
   winWidth 70
    size 1291
   BWwid 2000
    sideLobes 90
    cropT 0.0142857
    freqDrift 35
    freqFloor 70
   ampFloor -90
```

Loris Sound

LorisSound is a subclass of sound that has before/after versions of a sound and methods for managing the various envelopes and analysis behaviors.

LorisEditor GUI

The LorisEditor GUI has 4 main panes: 2 for sound views, a function view, and a spectrum view. Depending on the application, these might be an original and a resynthesized sound (for tuning analysis parameters), or two different sounds (for morphing). The menu bar at the top of the editor view contains items for all the important Loris functions.

CSL Models in Siren

The CSL unit generator models are in the Siren.CSL namespace. the CSLGraph class represents an instrument or DSP patch in the same sense that a CSL instrument object does. There are numerous example methods in the CSLGraph class.

To oest a CSL graph; play a simple note
```
[CSLGraph testSimpleGraph] d
```

Test FM
```
[CSLGraph testFM]
```

Test processors: filtered noise
```
[CSLGraph testStaticFilter]
[CSLGraph testDynamicFilter]
```

Play a sound sample
```
[CSLGraph testSoundFile]
```

Play 25 oscillators with random-walk freq and position.
```
[CSLGraph testOscillatorBank]
```

If you have a MIDI fader box, try the following demo
```
[CSLGraph testMIDIOscillatorBank]
```

Data Load/Store and the Paleo Database

Siren supports several mechanisms for object persistency. First of all, the Smalltalk virtual image facility allows one to save Siren data in a convenient database format (the Smalltalk virtual image snapshot file). The SirenSession class shared dictionaries hold Sounds, EventLists, and Voices in the virtual image, so that they are saved across snapshots. There's a utility method to load the user's sound and score data into the image, and one to flush the shared storage.

```
[SirenSession loadDemoData]      "load all your data into Smalltalk"
[SirenSession openTransport]     "See the sound and score menus"
"EventScheduler interrupt; flush"
"EventScheduler release"
[SirenSession initialize]        "flush everything"
```

Now use the pop-up menus in the transport view to browse and select sounds, scores, timers, etc.

Siren Native Format and BOSS

As described in the sections above, Siren can read and write a variety of standard multimedia file formats. Where we require features not present in other formats we also use several Siren-specific file formats, for example to store sets of envelope functions as breakpoint envelopes in a single file -- Siren's ".env" files supported by the Function class and used by the Loris package.

Siren also defines a new package file format called ".s7" files. These are folders (named xxx.s7) that contain various kinds of files (sound, SDIF, envelope, LPC data, etc.) that are to be treated as a group. This supports, e.g., versioning of Loris analysis/resynthesis files. The code that implements this format is in the class SirenUtility.

Paleo SMS (Squeak MinneStore)

Note: The paleo test data is not included with this release, and the examples below are stale.

MinneStore is a smalltalk native object-oriented database.

Load Event Lists from MIDI files

```
[   | num fn el |
   Cursor wait showWhile:
      [1 to: 100 do: [ :ind |
         num := ind asZeroFilledString: 3.
          fn := SirenUtility scoreDir, 'Scarlatti/K', num, '.MID'.
          (FileDirectory root fileExists: fn)
            ifTrue: [el := (MIDIFileReader scoreFromFileNamed: fn) asEventList.
                el at: #name put: ("ScarlattiK", num) asSymbol.
                el at: #composer put: "Domenico Scarlatti".
                el at: #instrumentation put: #harpsichord.
                el at: #style put: #Baroque.
                 Siren eventLists at: ("ScarlattiK", num) asSymbol put: el]]]]
```

The CREATE Real-time Applications Manager

CRAM is a distributed processing environment (DPE) database and management tool. For more information, see http://create.ucsb.edu/CRAM. CRAM assumes the existence of a Postgress-SQL database with network and application tables as described in the documentation. CRAM applications are generally written in C++ or SuperCollider. The CRAM manager is in Smaltalk.

NodeSocketInterface Examples

Try pinging the CRAM node manager on the local host.
```
NodeSocketInterface pingExample
NodeSocketInterface errorExample
NodeSocketInterface nameExample
```

Creating and starting a clock service via the CRAM node manager on the local host.
```
NodeSocketInterface createServiceExample
```

Try pinging the emergency port of the node manager on the local host.
```
NodeSocketInterface emergencyPingExample
```

Restart the node manager
```
NodeSocketInterface emergencyRestartExample
```

Try getting the log text of the node manager on the local host.
```
NodeSocketInterface getLogExample
```

Try getting the tail of the log text.
```
NodeSocketInterface getLogTailExample
```

Try creating and starting 2 clock services.
```
NodeSocketInterface handleServicesExample
```

Stop all running services.
```
NodeSocketInterface killServicesExample
```

Get the list of service names from the CRAM node manager on the local host.
```
NodeSocketInterface listServicesExample
```

Try getting the run-time statistics from the CRAM node manager on the local host.
```
NodeSocketInterface statisticsExample
NodeSocketInterface benchmarkExample
```

Database

Basic test using PostgresSql -- dump the machines table to the Transcript
```
DPE_Util test0
```

Manager GUI

CRAM.ManagerGUI open

Extended Siren Examples

There are a number of additional files with example code from my compositions
See the files in the StaleCode folder of the Siren 7.5 release, especially MusicApps-Pieces.st.

There is also a 15-minute sampler/tour of my music and the tools used to make it (almost all of them in Smalltalk) on my label's web site at
```
http://HeavenEverywhere.com/RitualAndMemory/Tour
```

Tools Used for Individual Pieces

Requiem Aeternam dona Eis (1985-4): DoubleTalk PetriNet simulation generated score files for cmusic.

Day (1986-7): EventGenerators + interactive interfaces for simlpe MIDI sequencing.

Kombination XI (1988-90): TrTree-based speech editors and prosody morphing.

All Gates are Open (1991-3): Vocoder scripting in Makefiles generated from a speech database.

Sensing/Speaking Space (1999): Segmented phonemes in 8S speech database used for speaker textures generated by SuperCollider.

Four Magic Sentences (2000): as above, + 12-tone rows as pitch classes.

Leur Songe de la Paix (2002-3): 8S database + drone lists + Morse code layer.

Eternal Dream (2003-5): Percussion event generators via sample libraries.

Jerusalem's Secrets (2005-6): EventGenerators for samlpe patterns.

Ora penso invece che il mondo... (2006): Serial methods dumped via MIDI to Sibelius notation package for (human) string quartet.

Works in Progress: CSL/Loris in event lists

Your Basic Siren Demo Script

To use this demo script, read through the text selecting the blocks enclosed in square brackets. The single character after the close-square-bracket (d,p, i, or db) denotes whether you should "do," "print," "inspect," or "debug" the block. (Typically, CTRL-D means do-it, CTRL-P means print-it, and CTRL-Q means inspect-it.) To look at the code for the complex examples below, simply "debug-it" and single-step into the demo method.

For the function and event list examples below, note that the "open" method edits the receiver by default; if you hold down while executing it, it will play the receiver, and if you hold down is will inspect the receiver.

If you're new to reading Smalltalk, look at the workbook section on "Learning to Read Smalltalk." If you're setting up Siren for the first time, see the section above on "Siren Set-up and Testing" and make certain you know how to use the Siren Utility panel and the Siren Transport view.

Set-up

Configure and test the MIDI and sound I/O drivers using the utility panel.

```
[SirenSession openUtility] d
```

Load the test data with the "Load All" button in the utility pane.
You can browse the test data with the left-side menu buttons in the transport panel, do,

```
[SirenSession openTransport] d
```

See also the section above on "Siren Set-up."

MusicMagnitudes

Print these to see what kinds of music magnitude representations and operations are supported.

```
[440 Hz asSymbol] p       "--> 'a3' pitch"
[(1/4 beat) asMsec] p       "--> 250 msec"
[#mf ampl asMIDI] p       "--> 70 vel"
[-16 dB asRatio value] p    "--> 0.158489"

['a4' pitch asMIDI] p
[('a4' pitch + 100 Hz) asMIDI] p
[('a4' pitch + 100 Hz) asFracMIDI] p
['mp' ampl + 3 dB] p
[('mp' ampl + 3 dB) asMIDI] p
[(1/2 beat) + 100 msec] p
```

Pitch expressions ("N" is short-hand for PitchClass)

```
[N C sharp sharp] p
[N C augmentedFourth] p
[N C diminishedThirteenth] p
[N do flat intervalBetween: N sol] p
[PitchClass D flat melodicMinorScale notes] p
[(PitchClassChord new fromString: 'C aug9 dim5') notes] p
[(HungarianMinor root: N fa) asPitchesInOctave: 2] p
```

Event Creation Messages

Create a `generic' event using a class instance creation message.

```
MusicEvent duration: 1/4 pitch: 'c3' ampl: 'mf'
```

Create one with added properties.
```
(MusicEvent dur: 1/4 pitch: 'c3') color: #green; accent: #sfz
```

Terse format: concatenation (with ',') of music magnitudes
```
[440 Hz, (1/4 beat), 44 dB] i
(#c4 pitch, 0.21 sec, 64 velocity) voice: IOVoice default
```

Event Lists

Verbose form using a class instance creation message;
a chord is simply a set of events at the same time.

```
(EventList newNamed: #Chord1)
    add: ((1/2 beat), 'd3' pitch, 'mf' ampl) at: 0;
    add: ((1/2 beat), 'fs3' pitch, 'mf' ampl) at: 0;
    add: ((1/2 beat), 'a4' pitch, 'mf' ampl) at: 0
```

Play a scale created with a class message.
```
[(EventList scaleExampleFrom: 48 to: 60 in: 1500) open] d
```

Create 64 random events with parameters in the given ranges, play it on the default output voice, or edit it.
```
[(EventList randomExample: 64          "make 64 notes"
   from: ((#duration: -> (50 to: 200)),      "duration range in msec"
         (#pitch: -> (36 to: 60)),         "pitch range in MIDI keys"
         (#ampl: -> (48 to: 120)),      "amplitude range in MIDI velocities"
         (#voice: -> (1 to: 1)))         "play all on voice 1"
   ) open] d
```

Create an event list of 20 notes with semi-random values and play it on a MIDI output voice.
```
[(EventList randomExample: 20) playOn: MIDIVoice default] d
```

Event lists don't have to have pitches at all, as in the word,
```
[EventList named: 'phrase1'
   fromSelectors: #(duration: loudness: phoneme:)          "3 parameters"
   values: (Array with: #(595 545 545 540 570 800 540)    "3 value arrays"
            with: #(0.8 0.4 0.5 0.3 0.2 0.7 0.1)
            with: #(#dun #kel #kam #mer #ge #sprae #che)).
(EventList named: 'phrase1') inspect]
```

Play two-voice "counterpoint" on the note list score file voices.
```
[ | vox list |
vox := CsoundVoice onFileNamed: 'test.cs'.
list := (EventList newNamed: #pRand)
      addAll: (EventList randomExample: 20);
      addAll: (EventList randomExample: 20).
vox play: list.
vox close.
(Filename named: 'test.cs') open] d
```

Here's another example of creating a simple melody
```
[(EventList named: 'melody'
   fromSelectors: #(pitch: duration: ampl:)
   values: (Array with: #(c d e f g)
            with: #(4 8 8 4 4) reciprocal
            with: #(1))) open] d
```

You can also create event lists with snippets of code such as the following whole-tone scale.
```
[ | elist |
elist := EventList newAnonymous.
1 to: 12 do:
   [ :index |
   elist add: (1/4 beat, (index * 2 + 36) key, 'mf' ampl)].
elist open ] d
```

Event lists can also be nested into arbitrary structures, as in the following group of four sub-groups
```
[ (EventList newNamed: 'Hierarchical/4Groups')
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (36 to: 40)), (#ampl: -> #(110)))) at: 0;
   add: (EventList randomExample: 8
      from: ((#duration: -> (60 to: 120)), (#pitch: -> (40 to: 44)), (#ampl: -> #(100)))) at: 1;
```

```
    add: (EventList randomExample: 8
       from: ((#duration: -> (60 to: 120)), (#pitch: -> (44 to: 48)), (#ampl: -> #(80)))) at: 2;
    add: (EventList randomExample: 8
       from: ((#duration: -> (60 to: 120)), (#pitch: -> (48 to: 52)), (#ampl: -> #(70)))) at: 3;
    open ] d
```

Smalltalk methods can also process event lists, as in this code to increase the durations of the last notes in each of the groups from the previous example.

```
  [ (EventList named: 'Hierarchical/4Groups') eventsDo:
     [ :sublist | | evnt |   "Remember: this is hierarchical, to the events are the sub-groups"
     evnt := sublist events last event.      "get the first note of each group"
     evnt duration: evnt duration * 8].      "multiply the duration by 4"
  (EventList named: #groups) open ] d
```

...or the following to take the scale and make it slow down

```
  [ | elist |
  elist := EventList scaleExampleFrom: 60 to: 36 in: 1500.
  1 to: elist size do:
     [ :index | | assoc |
     assoc := elist events at: index.
     assoc key: (assoc key * (1 + (index / elist events size)))].
  elist open ] d
```

Storage and Persistency

```
  "SirenSession eventList: 'piece1/mvmnt1/part1' put: EventList new"
  "SirenSession eventList: 'piece1/mvmnt1/part1'"
```

Siren Scheduler

Reset

```
  [EventScheduler initialize]
```

Here's how to use the event scheduler explicitly.

```
  [EventScheduler instance addClient: (EventList randomExample: 20) in: (500 msec).
  EventScheduler instance run] d
```

Flush and close down the scheduler

```
  [EventScheduler instance interrupt; flush] d
```

Action events have arbitrary blocks of Smalltalk code as their "actions." This example creates a list of action events that flash random screen rectangles.

```
  [ActionEvent playExample] d
```

Complex Multimedia Example

```
  [ | el |
  el := (Cloud dur: 6          "Create a 6-second stochastic cloud"
     pitch: (48 to: 60)        "choose pitches in this range"
     ampl: (40 to: 70)         "choose amplitudes in this range"
     voice: #(1)            "leave the 1 nil for now"
     density: 5) eventList.    "play 5 notes per sec. and get the events"
  1 to: el events size do:     "Now plug different voices in to the events"
     [ :ind |               "ind is the counter"
     (el events at: ind) event voice:
        (ind odd          "alternate between two voices"
           ifTrue: [MIDIVoice default]
           ifFalse: [OSCVoice default])].
                    "add some animation events"
  el addAll: ActionEvent listExample.
  el open] d              "and play the merged event list"
```

Functions and Control

If you like thicker function plots, do this,

```
  [FunctionView lineWidth: 2]
```

Basic ramp up/down (linear and exponential flavors)

```
  [(LinearFunction from: #((0 0) (0.5 1) (1 0))) at: 0.25] p
  [(ExponentialFunction from: #((0 0 5) (0.5 1 -5) (1 0))) at: 0.25 ] p
```

ADSR-like envelopes

```
[(LinearFunction from: #((0 0) (0.1 1) (0.16 0.7) (0.8 0.4) (1 0))) edit]
[(ExponentialFunction from: #((0 0 5) (0.1 1 -3) (0.8 0.4 -2) (1 0))) edit]
```

Open a view with a linear envelope, an exponential envelope, a spline curve, and a sum-of-sines function

```
[FunctionView multiFunctionExample]
```

One can apply a function to any property of an event list, as in the example below, which makes a crescendo/decrescendo using an exponential triangle function.

```
[ | list fcn |
list := EventList newNamed: #test3.
(0 to: 4000 by: 100) do:    "4 seconds, 10 notes per second"
      [ :index |          "add the same note"
       list add: (MusicEvent dur: 100 pitch: 36 ampl: 120) at: index].
fcn := ExponentialFunction from: #((0 0.05 2) (0.5 1 -2) (1 0.05)).
list applyFunction: fcn to: #loudness.
list inspect] d
```

Send function data values out as regular OSC messages

```
[OSCVoice functionExample] db
```

Load a SHARC sample and create a wave table from it.

```
[(Function from: (((SHARCInstrument fromDir: 'tuba') samples at: #c3) asWavetable: 1024)) edit: 1024] d
```

EventGenerators

A cluster is the simplest event generator.

```
[(Cluster dur: 2.0
    pitchSet: #(48 50 52 54 56)
    ampl: 100
    voice: 1) open] d
```

Chord object can give you an event list.

```
[((Chord majorTetradOn: 'f4' inversion: 0) duration: 1.0) open] d
```

Create and play a simple drum roll--another 1-D event generator.

```
[((Roll length: 2000 rhythm: 50 note: 60) ampl: 80) open] d
```

Create and edit a low 6 second stochastic cloud with 5 events per second.

```
[ | c |
c := (Cloud dur: 6          "lasts 6 sec."
    pitch: (48 to: 60)      "with pitches in this range"
    ampl: (80 to: 120)      "and amplitudes in this range"
    voice: (1 to: 1)        "select from these voices"
    density: 5) eventList.  "play 5 notes per sec. and get the event list"
c open] d
```

Play a 6-second cloud that goes from low to high and soft to loud.

```
[(DynamicCloud dur: 6
    pitch: #((30 to: 44) (50 to: 50))   "given starting and ending selection ranges"
    ampl: #((20 to: 40) (90 to: 120))
    voice: (1 to: 4)
    density: 15) open] d
```

Select notes frmo a given scale

```
[(SelectionCloud dur: 4
    pitch: ((NeapolitanMinor root: N do) asPitchesInOctave: 4)
    ampl: #(80 40 120)
    voice: #(1)
    density: 12) open] d
```

Play a selection cloud that makes a transition from one triad to another.

```
[(DynamicSelectionCloud dur: 6
    pitch: #( #(48 50 52) #(72 74 76) )   "starting and ending pitch sets"
    ampl: #(60 80 120)
    voice: #(1)
    density: 12) open] d
```

The extended DynamicSelectionCloud uses a multi-part pitch set of the format (time -> chord) (time -> chord) ... as in the following example.

```
[ | score chords list |          "generate the tetrads from the selected scale; scramble the order"
chords := ((NeapolitanMinor root: N do) generateChordsPoly: 4 inOctave: 2) scrambled.
list := ?OrderedCollection new.
1 to: 7 do:
   [ :ind |                              "shift every other one up 2 octaves"
   ind even ifTrue: [list add: ((ind - 1) * 3 -> ((chords at: ind) collect: [ :no | no + 24]))]
      ifFalse: [list add: ((ind - 1) * 3 -> (chords at: ind))]].
score := (ExtDynamicSelectionCloud dur: 8    "now make a cloud from these"
   pitch: list
   ampl: 60
   voice: nil
   density: 10) eventList.
score eventsDo: [ :ev |                      "plug in the properties for FM"
   ev inst: '/i1/pn'.
   ev modIndex: 2.0.
   ev ratio: 1.02.
   ev pos: 0.0].
SirenSession eventList: 'EvGens/dsCloud1' put: score.
score open] d
```

Mark Lentczner's bell peals ring the changes.

```
[(Peal upon: #(60 65 68)) open] d
```

EventModifiers

One can apply functions to the properties of event lists, as in the following example, which creates a drum roll and applies a crescendo modifier to it.

```
[ | roll decresc |
roll := ((Roll length: 3000 rhythm: 150 note: 60) ampl: 120) eventList.
decresc := Swell new function: (ExponentialFunction from: #((0 1 4) (1 0))).
decresc applyTo: roll.
roll open]
```

MIDI Control

Open MIDI, play notes based on the mouse position (x --> dur; y --> pitch) until mouse down.

```
[MIDIPort testMouseMIDI] d
```

Demonstrate program change by setting up an organ instrument to play on.

```
[MIDIPort testProgramChange] d
```

Down-load a general MIDI patch for a 16-voice percussion ensemble.

```
[MIDIPort setupTunedPercussion. MIDIPort testAScale] d
```

Reset the GM map

```
[MIDIPort resetEnsemble]
```

Demonstrate control commands by playing a note and making a crescendo with the volume pedal.

```
[MIDIPort testControlContinuous] d
```

Demonstrate pitch-bend by playing two notes and bending them.

```
[MIDIPort testBend] d
```

ANO

```
[MIDIPort allNotesOff]
```

Close down and clean up.

```
[MIDIPort cleanUp]
```

Voices and I/O

```
[CsoundVoice randomExampleToFileAndEdit]
[SuperColliderVoice randomExampleToFileAndEdit]

[(EventList randomExample: 20) playOn: MIDIVoice default]
[OSCVoice midiScaleExample]
```

These use the CSL OSC servers

```
[OSCVoice fmExample1]
[OSCVoice sndExample1]
```

These examples loop endlessly, so you have to interrupt or flush the scheduler to stop them

```
[OSCVoice fmExample2.       5 wait.
OSCVoice sndExample2.       5 wait.
OSCVoice fmExample4]
 [EventScheduler flush]
```

As an example that mixes styles, the following expression plays a long low FM note and then uses Siren function objects to send continuous controls to make the note glissando down and pan from left to right.

```
[OSCVoice fmExample3]
```

The Siren Graphics Framework

Display rectangles in a display list view -- test zoom and scroll.

```
[DisplayList rectangleExample]
```

An alternative layout (which I prefer) places the zoom bars on the top and right. look at,

```
[DisplayListView open4SquareOn: (DisplayList rectanglesX: 2000 byY: 2000)]
```

Display random strings

```
[DisplayList stringExample]
```

Show the result of the IndentedListLayoutManager

```
[DisplayListView colorClassListExample]
[DisplayListView classTreeExample]
```

Music Notations

Open a sequence view on a random event list.

```
[TimeSequenceView randomExample] d
```

Try the picth-time layout

```
[PitchTimeView randomExample] d
[PitchTimeView openOnEventList: (EventList scaleExampleFrom: 48 to: 84 in: 5)] d
```

Open a pitch/time view on a *very long* 3-stream event list.

```
[PitchTimeView randomExampleLong] d
```

A more complete example is Hauer-Steffens notation, which has a clef and staff lines as in common-practise notation.

```
[HauerSteffensView randomExample] d
[(EventList scaleExampleFrom: 40 to: 66 in: 5) edit] d
```

Sound Views

Create and view some example sounds

```
[SoundView openOn: SampledSound sawtooth] d
[(SampledSound sweepDur: 10.0 rate: 44100 from: 10 to: 400 chans: 1) edit] d
```

Read in a sound from a file

```
[(SampledSound fromFile: 'unbelichtet.aiff') edit] d
```

Now try the external interface examples, then the SWIG Loris and CSL APIs.

Create a swept sine wave and take its fft.

```
[Spectrum sweepExample display] d
```

Read a file (T'ang dynasty speech) and show the spectrogram

```
[Spectrum fileExample display] db
```

Low-level sample processing: sum a sine and a sawtooth

```
[ | sin saw |
sin := SampledSound sineDur: 1.0 rate: 44100 freq: 10 chans: 1.
saw := SampledSound sawtoothDur: 1.0 rate: 44100 freq: 100 chans: 1.
 sin scaleBy: 0.8.
 saw scaleBy: 0.1.
 1 to: sin size do:          "loop to do vector math on sound samples"
    [ :index |
     sin at: index put: ((sin at: index) + (saw at: index))].
 sin edit] d
```

As a final example, one can apply a function to a sound as an envelope, as in this block,

```
[((SampledSound sineDur: 1.0 rate: 44100 freq: 220 chans: 1)
        scaledByFunction: (ExponentialFunction default)) edit] d
```

Using the Loris Interface and Tools

For normal usage, set-up with this

```
[Siren.Loris.Loris initializeModule] d
```

Test with this (prints the Loris version string)

```
[Siren.Loris.Loris version] p
```

To load a sound file and run it, saving the result as SDIF and opening a display (read this method to see how to use the Loris API)

```
[LorisSound example: '1.2a1.aiff' resolution: 70] db
[LorisSound fromSDIF: 'horns1c.sdif'] d
```

Using the CSL Models

Set-up

```
[Siren.CSL.CSL initializeModule] d
```

First test; print a random number

```
[Siren.CSL.CSL fRand    ] p
```

To oest a CSL graph; play a simple note

```
[CSLGraph testSimpleGraph] d
```

Test FM

```
[CSLGraph testFM]
```

Test processors: filtered noise

```
[CSLGraph testStaticFilter]
[CSLGraph testDynamicFilter]
```

Play a sound sample

```
[CSLGraph testSoundFile]
```

Play 25 oscillators with random-walk freq and position.

```
[CSLGraph testOscillatorBank]
```

If you have a MIDI fader box, try the following demo

```
[CSLGraph testMIDIOscillatorBank]
```

Building a Siren Image

To load Siren into a VisualWorks 7.5 virtual image, follow these steps.

Start VW 7.5

```
Load your favorite parcels (AT tools, DB, etc.)
Siren requires the following packages:
   Store/PostgreSQL, DLLCC, AdvancedTools, BOSS, XML Tools,
   HTTP and ComposedTextEditor from the standard release, and
   SmaCC and SWIG from the public Store repository
Load Siren.pcl
It should automatically do the next 2 steps, see the Transcript.
File in the MusicConstants.st file
   [((Net.HttpClient get: 'http://FASTLabInc.com/Siren/MusicConstants.st')
         contents readStream binary) fileIn]
Load the workbook browser from the BOSS file
   [(ListWorkBook new loadFromURL: 'http://FASTLabInc.com/Siren/Siren7.5_Workbook.bos') open]
```

By-hand init (see these methods for site tuning)

```
Siren.SoundFile initializeSoundFileFlags
Siren.SirenUtility initialize (or SirenUtility initializeSirenSTP)
```

Load optional L&F hacks

```
Left-hand scroll bars - BorderDecorationPolicy-initialize.st
Custom colors - MotifWidgetPolicy class-initializeDefaultGenericColors.st
```

To open list workbook by hand

ListWorkBook open
Use the Page/load_all menu item to load the workbook contents
from the BOSS file Siren7.5_Workbook.bos

See the appropriate Workbook pages for the following subsystems

Load and test the DLLCC External Interfaces (see the C makefiles and interface class defs)
LibSndFile - [LibSndFileInterface example1: 'unbelichtet.aiff']
PortMIDI - [PortMidiInterface testMIDI]
FFTW - [FFTWInterface example]
PortAudio - [PortAudioInterface example1]

Test the OSC I/O (assumes some OSC client)
OSCVoice midiScaleExample

Load and test the SWIG Interfaces
CSL
Siren.CSL.CSL initializeModule
CSLGraph testSimpleGraph
Loris
Siren.Loris.Loris initializeModule
Siren.Loris.Loris version
See examples in LorisSound

Set up a new changelist
make a snapshot...

Now for the bad news:
Undeclared inspect

Related Software

Siren is typically used in consort with a number of other packages; for more information, see the links below

CSL: the CREATE Signal Library: a C++ framework for building OSC-controllable synthesis/processing servers
http://create.ucsb.edu/CSL

Loris: a sound analysis/resynthesis package written by Kelly Fitz
http://sourceforge.net/projects/loris

SuperCollider: an efficient synthesis and processing language very similar to Smalltalk
http://supercollider.sourceforge.net/

CRAM: the CREATE Real-time Applications Manager: a tool for describing and managing complex distributed real-time software systems
http://create.ucsb.edu/CRAM

Occam: An OSC-to-MIDI translator
http://www.mat.ucsb.edu/~c.ramakr/illposed/occam.html

Macco: a MIDI-to-OSC convertor
http://www.create.ucsb.edu/CSL/Macco.zip

FMAK: the FASTLab Music analysis Kernel (a sibling of CSL)
http://fastlabinc.com/

libSndFile: http://www.mega-nerd.com/libsndfile

PortMidi: http://www.cs.cmu.edu/~music/portmusic

PortAudio: http://www.portaudio.com

FFTW: the fastest FFT in the west
http://www.fftw.org

Siren 7.5 Release Notes

Current Status

This is the V7.5 release of Siren on VisualWorks. (For simplicity, Siren version numbers parallel the versions of Smalltalk on which they run.)

Smoke: The Smoke representation is complete, see the event list and event generator examples.

Voices and Schedulers: The scheduler delivers acceptable real-time performance on a modern CPU. The real-time output via OSC or MIDI is suitable for use in performance.

Sound I/O: Sound file I/O is implemented for many kinds of sound files (AIFF, WAV, .snd, etc.) via the libSndFile interface. There is a PortAudio callback interface using DLLCC as well as an FFTW interface for the class Spectrum

MIDI/OSC: Ports and Voices exist for real-time control I/O.

There are DLLCC interfaces to CSL and Loris based on SWIG interfaces.

Graphics and GUI: The display list framework, Navigator MVC framework, and several MVC-based GUI tools are available; see the examples above.

Version History

Siren on Visualworks
- 7.5--CREATE, Santa Barbara, April, 2007
- 7.4--CREATE, Santa Barbara, Winter, 2006
- 7.3--CREATE, Santa Barbara, throughout 2005
- 7.2--CREATE, Santa Barbara, Fall, 2003
- 7.0--CREATE, Santa Barbara, Spring, 2001

Siren on Squeak
- 3.0--T. U. Berlin, Summer, 2000
- 2.2--CREATE, October/November, 1998
- 2.0--CREATE, April-June, 1998
- 1.3--CREATE, August/September, 1997
- 1.0--CREATE, December, 1996

MODE
- 2.0--(Topaz2) CNMAT/Berkeley, April - December, 1994
- 1.3--(Topaz) SICS/EMS, Stockholm, February - April, 1993
- 1.2--CCRMA/Stanford, July, 1991 - February, 1992
- 1.0--STEIM, Amsterdam, May/June 1990
- 0.8--CCRMA/Stanford, June, 1989
- 0.4--ParcPlace Systems, March, 1988

HyperScoreToolKit
- 1.0--Xerox PARC, October, 1986

DoubleTalk
- 1.0--PCS/Cadmus GmbH, September, 1985

Reference, Acknowledgments

The main Siren Web site is at http://FASTLabInc.com/Siren.
This outline is on-line at http://FASTLabInc.com/Siren/Doc.
To join the mailing list, see http://www.create.ucsb.edu/mailman/listinfo/Siren.

To read more about computer music, get a copy of Curtis Roads' "Computer Music Tutorial" (MIT Press).

To learn more about Smalltalk, see the several excellent on-line Smallatlk tutorials (just ask Google) e.g.,
- http://www.cincomsmalltalk.com/tutorials/version7/tutorial1/home.htm
- http://www.smalltalk.org
- http://www.squeak.org
- http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/smalltalk1.html
- http://daitanmarks.sourceforge.net/or/squeak/squeak_tutorial.html
- http://www.sra.co.jp/people/aoki/htmls/FreeDocumentsForSmalltalk.html

The Squeak CD-ROM (http://www.squeak.org/Download/SqueakCD) has a whole collection of good Smalltalk tutorials.

Acknowledgments

Siren incorporates the work of many people who have contributed ideas and/or code; it would be most unfair not to acknowledge them here. They include:

Paul Alderman
Alberto de Campo
Roger Dannenberg
Lounette Dyer
Adrian Freed
Guy Garnett
Kurt Hebel
Frode Holm
Helge Horch
Dan Ingalls
Craig Latta
David Leibs
Mark Lentczner
Hitoshi Katta
Alex Kouznetsov
John Maloney
James McCartney
Hans-Martin Mosner
Luciano Notarfrancesco
Danny Oppenheim
Nicola Orio
Francois Pachet
Andreas Raab
Chandrasekhar Ramakrishnan
Pierre Roy
Carla Scaletti
Bill Schottstaedt
John Tangney
Bill Walker

I must also here acknowledge the generous support of my employers and the academic institutions where this software (and its predecessors) was developed, including CMRS@Mozarteum/Salzburg (1980-85), PCS/Cadmus GmbH in Munich (1983-6), Xerox PARC (1986-89), ParcPlace Systems, Inc. (1988-94), CCRMA@Stanford (1986-92), the STEIM Foundation in Amsterdam (1990-91), the Swedish Institute for Computer Science (1992-93), CMNAT@UC Berkeley (1993-95), CREATE@UC Santa Barbara (1996-2006), and the Technical University of Berlin (2000).

A Note on Reading the Smalltalk-80 Programming Language

(stp - 12/91)

Smalltalk-80 is a message-passing object-oriented programming language where expressions are built to be read as English sentences. The language is based on the concepts of objects (software modules that consist of state [data] and behavior [procedures]), that send messages among each other. All data is organized into objects, and all functionality is described as the behaviors of objects. A Smalltalk- 80 application is a network of objects.

Objects that have similar state and behaviors are grouped into classes (e.g., the class of all whole numbers [integers], or the class of all 2-dimensional Cartesian points). The classes themselves are arranged into a tree-like sub/superclass hierarchy with inheritance of state and behavior from a superclass to its subclasses (e.g., the class of integers might be defined as a subclass of the class of numbers).

Names and identifiers are separated by white space and are often composed of a concatenated noun phrase written with embedded upper-case letters, e.g., anEvent or MusicMagnitude.

In a Smalltalk-80 expression, the noun (i.e., the receiver object), comes first, followed by the message; e.g., to ask the size of an array, one sends it the message "size," as in [anArray size].

If the message has any arguments, they are separated from the message keywords in that the keywords all end in ":", so to index the first element of the array, one writes [anArray at: 1]; to set it to zero, use [anArray at: 1 put: 0].

Binary operators are allowed for mathematical (+, *, etc.) and logic (>, ~=, etc) operations.

Expressions can be nested using parentheses (i.e., evaluate what's inside the inner-most parentheses first), and fancy expression constructs and control structures are possible.

Expression examples

```
Unary messages - [receiver message]
   anArray size
   Date today            "message to a class"
```

```
    x negated

  Binary messages - [receiver operation operand]
    3 + 4.2215
    x > 7              "answers true or false"
    (a + b) * (c + d)

  Keyword messages - [receiver keyword: operand (keyword2: message2)]
    anArray at: 1              "get an item from an array"
    anArray at: 1 put: 7       "assign into an array"
    aVisualItem displayOn: aMedium at: aPoint clip: aClipRect mask: aMask
  (messages called at:put: or displayOn:at:clip:mask:)

Unary expressions can be concatenated, as in
  Date today weekday size odd printString

which answers the string "true" or "false," as though it were written,
  ((((Date today) weekday) size) odd) printString

Unary messages bind more strongly than binary messages, as in,
  3 negated * 4 sqrt       "answers -6 "
this is read as
  (3 negated) * (4 sqrt)

Keywords bind the weakest, so that the expressions,
  Transcript show: aNumber printString, ' is not equal to ', otherNumber printString.
  anArray at: index + 1 put: 'hello ' , ' world'.
are to be read as,
  Transcript show: ((aNumber printString), ' is not equal to ', (otherNumber printString)).
  anArray at: (index + 1) put: ('hello ' , ' world').

A single expression can contain at most one keyword message, i.e., the compiler always tries to build the longest
composed keyword message it can find in an expression. For example, to copy an element from one position in an
array to another, we would have to parenthesize the 2 keyword messages, as follows,
  anArray at: 1 put: (anArray at: 2)
because without the parentheses, the message would be interpreted as at:put:at: (which is probably not implemented).

When several messages are to be sent to the same receiver, and the return values are ignored, one can create a
message cascade" by separating the messages with semi-colons instead of periods, and not repeating the receiver, as
in,
  Transcript show: name printString.
  Transcript cr.
which can be written as a cascade or 2 messages sent to Transcript,
  Transcript show: name printString ; cr.
```

Comments

Double-quotes delineate comments in Smalltalk-80 (e.g., "a comment"); single-quotes are used for string objects (e.g., 'a string').

Naming

Most variable and message names start with lower-case letters and may contain embedded upper-case latters (camelCase) as in the examples above.

Global-scope objects such as namespaces, shared variables and classes are typically capitalized, as in Date, OrderedCollection, or Siren.Sound (a class name in a namespace).

Names for temporary variables are declared between vertical bars (e.g., | varName1 varName2 |).

Symbols are special strings that are stored in a table so as to be unique; they are written with the hash-mark "#" as in #blue, meaning "the symbol blue."

Immediate Object Formats

```
  Integer        7          244361990863443121
  Real       3.1415926      2.8879044346233
  Character     $a           $.
  String       'hi'         'longer string'
  Symbol       #hi          #SymbolicValue
```

```
List (Collection)   #(a b c)
Association       ( a -> b)            "key/value tuple"
Dictionary (Map)   (a -> b), (c -> d)      "like a hash map"

Logical constants       true, false, nil
```

Control Structures

Smalltalk supports deferred execution in the form of closures or anonymous functions called blocks; they are written in square brackets "[...]" and can have arguments and/or local temporary variables.

The up-arrow or caret (^) is used to return values (objects) from within blocks.

A block that takes two arguments and returns their sum would look like:

```
[ "arguments" :x :y | "body" ^(x + y)].
```

Since Boolean objects and blocks are supported, it's easy to define the standard program logical control flow operations such as if/then/else or do/while. Here are examlpes in Smalltalk

```
(x < 0) ifTrue: [x := x negated]          "like if (x < 0) { x = 0 - x; };"
(x < 0) ifTrue: [x := x negated]          "like if (x < 0) { } else { }"
      ifFalse: [x := x - 1]
```

For loop: use x to: y do: [block-with-1-argument]

```
1 to: 10 do: [ :index | Transcript show: index printString; cr]
```

While loop

```
[x > 0] whileTrue: [Transcript show x printString; cr. x := x - 1]
```

Collection iteration

```
#(1 3 5 7 9) do: [ :item | "block " ]
aSet select: [ :item | item > 0 ]
```

The format of a method; the browser uses a standard code template for new methods

```
methodName: argument
   "Comment stating purpose and return value."

   | tempVarName |
    method body statements.
    ^self          "some return value"
```

Short-hand and Important Messages

#x - the symbol with the name x

#(a b c) - an Array with members a, b, & c

x -> y - an Association (x = key, y = value)

x @ y - a Point with X = x and Y = y

The Protocol of All Objects

There are many methods implemented in class Object that are understood by everyong in the system (i.e., you can send them to self in any method).

```
basicAt:, basicAt:Put: - encapsulation-breaking low-level access
halt - a breakpoint to get into a debugger
inspect - look inside the receiver
printString - pretty-print the receiver
storeOn: - serialize the receiver on a stream
```

All classes understand some useful instance-accessing messages

```
instanceCount - how many of me are there?
allInstances - get them all
someInstance - pick one
```

The Class Library

Smalltalk includes a rich class library, the main abstractions of which are listed below.

Magnitudes - understand comparison operators

Subclasses like ArithmeticValue, Point, Number, SmallInteger, etc.
(arbitrary-precision integers, fractions, meta-numbers)

Collections - aggregate objects such as arrays and hash maps
Set, Dictionary, OrderedCollection, SortedCollection, Array
Understand size includes:, do: collect: select:

Streams - read/write streaming I/O, used for files, sockets, etc.
Understand next (read something) and nextPut: (write)

Model/View/Controller - classes for the MVC design pattern used in GUIs

Networking and Protocols - Support for formatted streams

UI Support - ApplicationModel, Dialog

Compilers - Compiler tools

Programming

Smalltalk programs are organized as the behaviors (methods) of classes of objects. To program a graphical application, for example, one might start by adding new methods to the point and 3-D point classes for graphical transformations, and build a family of "smart" visible display objects that know how to present themselves in interesting ways.

Classes are described as being abstract or concrete depending on whether they are meant as models for refinement within a framework, or for reuse "off the shelf" as in the elements in a tool kit.

Inheritance means that classes of objects are related in hierarchies (i.e., abstract and concrete classes related in trees), where they share their methods and state variables. This means that the class of 3-D points only has to add the z- coordinate and a few new methods when it's defined as being a specialization (subclass) of the 2-D point class.

Polymorphism means that many kinds of objects respond to the same message with their own methods to carry out related behavior. Examples are the "play" message being handled by event lists in terms of being passed on to their component events in real-time, or of displayable objects and windows all having methods for the "display" message.

Inheritance and polymorphism mean that one reads Smalltalk-80 programs by learning the basic protocol (messages/methods) of the abstract classes first; this gives one the feel for the basic behaviors of the systems's objects and applications.

To actually work in Smalltalk, one generally writes code in browsers, then uses inspectors to create and manipulate objects during testing. One can also write tests and demonstrations as class example methods. There are many browser and debugger features that encourage extreme and exploratory programming. Code generally lives in a database and can be easily shared through the various Internet-based code repositories such as Store or Envy.

For more info, there are several excellent on-line Smallatlk tutorials (just ask Google), see the reference section of this workbook. The Squeak CD-ROM (http://www.squeak.org/Download/SqueakCD) has a whole collection of good Smalltalk tutorials.

Known Bugs in Siren

Kernel Classes
Stable
ConditionalDuration is not finished

Support Classes
Stale DB code still present
Some unused code in class examples
There are a few places in the code where UNIX-style file names are assumed.

Sound and functions
Sound cut/paste using composite sounds needs to be fixed

IO and Voices
MIDI file reader only reads version 2 format files
Call-back based Sound IO (SmartAudioPort) is still buggy -- output clicks, recording also buggy
(I use file-based sound I/O or CSL)

Graphics and GUI
The GUI views are "demo-quality" -- most are output-only

Polyline display doesn't zoom
Double redisplay on zoom of pitch-time views
Staff in HS view is screwy

List of ThingsToDo

Port mixer classes
Port harmonic analysis tool
Complete the EventListTreeEditor
CNM View
Sound fromFunction:
FcnEvent delta + interval examples
ConditionalDuration

DLView adornment switches
White/pink noise generators
Dynamic panel view with snd/fcn sub-panes
Finish CSL patch class: envelopes, triggering, use with CSLVoice

---