The Siren Workbook

Outline

Go to the Siren Home Page

Introduction to the Siren Workbook

What is this?

This document is the Siren Version 7.2 user's guide. This manual is available as a web site (http://create.ucsb.edu/Siren/Doc), a PDF file for printing (http://create.ucsb.edu/Siren/7.2.Workbook.pdf), or on-line in a workspace outline view within Smalltalk (the best way to read it).

Throughout this document, there are many executable Smalltalk expressions, generally enclosed in square brackets, as in
```
[ some code]
```
If you are reading this text within a Smalltalk run-time system, 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 posses 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.

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 250 Smalltalk classes for building musical applications. The current version of Siren (7.2) works on VisualWorks Smalltalk (available for free for non-commercial use, see http://www.cincom.com/smalltalk) 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://create.ucsb.edu/Siren.

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, and MIDI I/O);
user interface components for musical applications
(UI framework, tools, and widgets); and
several built-in applications
(editors and browsers for Smoke objects).

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?

Various versions and components of Siren"s predecessors (DoubleTalk, The HyperScore ToolKit, and the MODE) are documented in several places:
-- "The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology" (S. T. Pope, ed. MIT Press, 1991);
-- "The Interim DynaPiano" in "Computer Music Journal" 16:3, Fall, 1992 (also on the CMJ Web site);
-- "Musical Signal Processing" (C. Roads, S. T. Pope, G. DePoli, and A. Piccialli, eds. Swets & Zeitlinger, 1997);
-- "Squeak: Open Personal Computing and Multimedia" (Mark Guzdial and Kim Rose, eds, Prentice-Hall, 2002);
-- Proceedings of the 1986, 1987, 1989, 1991, 1992, 1994, 1996, 1997, 2003 International Computer Music Conferences (ICMCs); and
-- in several documents accessible via the Web page http://create.ucsb.edu/~stp/publs.html.

There are more MODE- and Smoke-related documents (including several of the above references) in the directory ftp://ftp.create.ucsb.edu/pub/Smalltalk/Music/Doc.

The official Siren home page is http://create.ucsb.edu/Siren.

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). The most recent advances incorporated in Siren 7.2 stem from from the realizations of "Eternal Dream" (2002) and "Leur Songe de la Paix" (2003). In each of these cases, some amount of effort was spent--after the completion of the composition--to make the tools more general-purpose.

A Note on Support

Siren is not supported, and most new features and extensions are to be found in the VisualWorks version of Siren.

Getting Started with Siren

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

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 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.

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.

Executing code: do-it and print-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 toe 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 ] db
```

Setting up Siren

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

```
[SirenUtility open]
```

You can see the system configuration and test the MIDI, sound file, and sound IO here.

The SirenUtility and the Workbook are available as menu items in the Launcher's Tools menu.

Setting up the external interfaces

Siren uses several external interfaces (based on the DLLCC framework) for access to external data and I/O. The Smalltalk code for

these interfaces is in the category Music-External.

Sound file read/write -- libSndFile -- see http://www.zip.com.au/~erikd/libsndfile
Streaming sound I/O -- PortAudio -- see http://www.portaudio.com
Streaming MIDI I/O -- PortMidi -- see http://www-2.cs.cmu.edu/~music/portmusic

You need to have these libraries installed, and compile and link the DLLCC interface libraries in the subdirectory DLLCC for sound or MIDI I/O to work with Siren. All of this has been tested on Mac OSX only. There are pre-compiled versions of the required libraries on the CSL web site at CREATE; look at http://create.ucsb.edu/CSL.

Testing sound file I/O
This will print diagnostics and a few rows of sample values to the Transcript.
(Edit the file name to point to a valid sound file.)
[LibSndFileInterface example1: 'Data/a.snd']

Testing streaming sound I/O (this will report to the Transcript)
[PortAudioInterface example1]

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

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. The basic Squeak built-in MIDI can be used, but is limited to toy examples. There are special virtual machine extensions and drivers for the Macintosh and MS-Windows platforms.

Smalltalk Options

There are several configurable parts to Siren.

Class Siren 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 "TestData" and is a sub-folder of the folder where the virtual image is executing. There are methods in class Siren to change this.

Siren House-keeping

To clear out temp. event lists, use,
[SirenUtility flushTempEventLists]
or to flush all,
[SirenUtility 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's any cruft,
[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).

These subsystems use a number of design patterns, implementing visitors, adaptors, double-dispatching, MVC, and others.

The Siren Class Categories

Music-Models-Representation -- basic MM models
Music-Models-Implementation -- MM implementation classes
Music-Events -- Events, EventLists
Music-EventGenerators -- EGs
Music-EventModifiers -- EMs
Music-Functions -- functions
Music-Sound -- Sound classes
Music-Support -- Scheduler, Utility

MusicIO-External -- External interfaces to PortAudio, PortMIDI, 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 editor
MusicUI-Layout -- Layout managers
MusicUI-Sound -- Sound view

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
      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
        MixElement
```

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
            MIDIEcho
          Mix -- output clipped rate channels
```

Functions

```
  AbstractEvent -- properties
    DurationEvent -- duration
      Function -- data range domain
        FourierSummation -- myForm myArray
        LinearFunction
          ExponentialFunction
          SplineFunction -- linSeg
        Sound
          GranularSound -- grains
          StoredSound -- samplesInMemory firstIndex changed
            FloatSound
```

```
        Mu8Sound
        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 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

Pitches
  HertzPitch -- 440.0 hz
  MIDIPitch -- 60 pitch -- can be non-integer for microtonal tunings (use #asFracMIDI)
  SymbolicPitch -- 'c#3' pitch -- can have a remainder for microtonal tunings
  RatioPitch -- 11/9 of: anotherPitch -- used for fraction-oriented tunings

Durations
  SecondDuration -- 1 sec
  MSecondDuration -- 100 msec
  RatioDuration -- 1/4 beat

ConditionalDuration -- until: [:t | block]

Amplitude/Loudness Objects
DBLoudness -- -3 dB -- can be relative to 0 dB or positive-valued
RatioLoudness -- 0.7071 ampl
SymbolicLoudness -- 'fff' ampl
MIDIVelocity -- 96 vel

Other Music Magnitudes
OrdinalMagnitudes -- have order but no explicit value
PField -- name/slot/value -- used for note lists

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"
60 pitch          "a MIDIPtch"
250 msec          "a MSecondDuration"
1/4 beat          "a RatioDuration"
```

MusicMagnitude Coercion Examples

```
440 Hz asSymbol     "--> 'a3' pitch"
(1/4 beat) asMsec   "--> 250 msec"
#mf ampl asMIDI     "--> 70 vel"
```

Duration Coercion Example--create a 1/8 beat duration and coerce it into msec, 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 of command key to "do it.")

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

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.

Smoke Events

All musical structures in Smoke--from micro-sound components of a note, to entire cmopositions--are represented via 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 get/set methods: (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).

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
MusicEvent -- adds pitch 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

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--and Smalltalk-80 collection iteration and enumeration messages such as (anEventList select: [someBlock])--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]
```

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))]
```

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 event association.

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]
```

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 msec)

```
[(EventList scaleExampleFrom: 48 to: 60 in: 1500) play]
```

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)))) play]      "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: -> (150 to: 400)),
        (#pitch: -> (36 to: 60)),
        (#ampl: -> (48 to: 120)),
         (#voice: -> #(organ1 flute2 clarinet bassoon1 marimba bass1)))))]
```

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

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

```
[EventList scaleExample2 inspect]
```

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)) play]
```

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 play ]
```

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;
   play ]
```

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) play ]
```

...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 play ]
```

Storage and Utilities

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

```
[SirenUtility eventLists inspect]
```

If you create an event list with a name that contains the character '/', then it is assumed to be in a subdirectory of the top-level event list dictionary, as in the example abive 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.

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

```
[SirenUtility flushTempEventLists]
```

or to flush all,

```
[SirenUtility flushAllEventLists]
```

Inspect a dictionary of all known event lists.

```
[SirenUtility eventLists inspect]
```

To read in a stored file, simply,

```
[(FileStream fileNamed: 'events.st') fileIn]
```

Load all event lists (.ev, .midi, and .gio files), from the given directory.

```
[EventList loadDirectory: Siren scoreDir]
```

Siren Functions

There are several classes that support functions of 1 variable such as envelopes or waveforms. These objects can be created (e.g.,) using linear or exponential interpolation between 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 atX: to get at their values.

Examples

Basic ramp up/down

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

ADSR-line envelopes

```
[(LinearFunction from: #((0 0) (0.1 1) (0.2 0.7) (0.9 0.5) (1 0))) edit]
[(ExponentialFunction from: #((0 0 5) (0.1 1 -3) (0.8 0.5 -2) (1 0))) edit]
[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: 128 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

Apply a function to a property of an event list--make a crescendo/decrescendo

```
[ | 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 2) (0.5 1 -2) (1 0)).
list applyFunction: fcn to: #loudness.
list play ]
```

Spectra and Signal Analysis (Not yet ported)

Create a swept sine wave and take its fft.

```
[Display restoreAfter: [Spectrum sweepExample]]
```

Read a file ("unbelichtet," etc. in German) and show the spectrogram
```
[Display restoreAfter: [Spectrum fileExample]]
```

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.

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 1991 ICMC paper.

Examples

Clusters and Chords are simple one-dimensional event generators.

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

[((Chord majorTetradOn: 'f4' inversion: 0) duration: 1.0) edit]
[((Chord majorTetradOn: 'f4' inversion: 1) duration: 1.0) play]
```

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

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 play
"c edit"]
```

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) edit]

[(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: 6) play]
```

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) play "edit"]
```

By obvious extension, andynamic selection cloud allows one to specify the start and finish selection sets. 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) play]

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

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

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

[ | 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 play]
```

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

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

```
[ | roll rub |
roll := ((Roll length: 5000 rhythm: 150 note: 60) ampl: 80) eventList.
rub := Rubato new function: (LinearFunction from: #((0 1) (1 0.5))).
rub applyTo: roll.
roll edit]
```

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) that are assumed to be able to do something in response to the (scheduleAt: aTime) message. The return value from this message is assumed to be the delay 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 global EventScheduler instance can be accessed by a class message (schedule) to class Siren; 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 follow.

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

[EventScheduler instance addClient: ActionEvent listExample in: (1000 msec); run]

[EventScheduler instance isRunning]
[EventScheduler instance interrupt]
[EventScheduler instance flush]
```

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]
```

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 or OSC 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 that takes the properties of an event and creates a statement of command for some output format. this allows the user to customize the voices at run-time.

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 default]
```

Read MIDI, turn it into a Siren event list, and play it. (BROKEN in 7.2)

```
[((MIDIFileReader scoreFromFileNamed: 'BWV775.MID') asEventList) play]
```

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"
```

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 host platform's native MIDI driver (in our case, 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 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.

MIDI Tests and Examples

Basic Tests

"Try to open and close the MIDI port (report to transcript)."
[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]

"Close down and clean up."
[MIDIPort cleanUp]

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]

"Reset the GM map"
[MIDIPort resetEnsemble]

MIDI Input

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

"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 30 seconds,
then turn off. The default update: method just dumps the MIDI packet into the transcript;
customize this by writing your own update: method."
[MIDIPort dumpExample]

"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]
```

Utilities

```
"ANO"
   [MIDIPort allNotesOff]

"Close down and clean up."
   [MIDIPort cleanUp]
```

Siren and OpenSoundControl

Siren includes an output voice that generates the CNMAT OpenSoundControl (http://www.cnmat.berkeley.edu/OpenSoundControl) protocol, which is sent out over UDP network packets. We generally use SuperCollider of CSL to create OSC sound synthesis servers, and then control them with messages sent out form Siren. For simple debugging, Chandrasekhar Ramakrishnan wrote Occam (http://www.mat.ucsb.edu/~c.ramakr/illposed/occam.html) a stand-alone OSC-to-MIDI convertor for Mac OS X.

The built-in examples demonstrate using OSC with the Occam convertor, to test OSC output using a MIDI synthesizer.

```
   [OSCVoice midiScaleExample]
```

Sound Objects

Siren includes objects that support sampled sound synthesis, recording, processing, and playback. In Siren, sound is a function, meaning that it has the semantics of a single-valued function of time.

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: 20 width: 0.0005 chans: 1) edit]

View a sawtooth waveform
   [SoundView openOn: SampledSound sawtooth]
```

Read in a sound from a file
```
[(SampledSound fromFile: 'Data/unbelichtet.aiff') edit]
```

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 fuile using your favitors snd file editor.

Read various file formats
```
[(SampledSound fromFile: 'Data/unbelichtet.aiff') edit]
[(SampledSound fromFile: 'Data/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. An instance of SoundPort communicates with the external driver.

This example plays a 3-second sine wave sweep
```
[PortAudioPort test2]

SampledSound sweepExample play
```

The Siren Graphics Framework

NB: The Siren graphics and display framework is for demo purposes only. Most of the views/editors are support output only (no complex editing), and may even have bugs.

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: 2000 byY: 2000)]
```

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]
```

Display *lots* of random poly-lines in a very large space (zoom out to see it all).
```
[DisplayList polylineExampleHuge]
```

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

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: 10000)]
```

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]
[(EventList scaleExampleFrom: 36 to: 48 in: 3000) 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.

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

Data Load/Store and the Paleo Database

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

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]]]]

Siren eventLists explore
EventList someInstance
EventList instanceCount
EventList allInstances
```

Loading Sound Files

```
[   | dir fn |
    dir := Siren soundDir, "SimpleT/justin_b/".
   Transcript cr.
   10 to: 14 do: [ :ind |
       fn := (FileDirectory on: dir) fileNamesMatching: (ind printString, "*").
      fn isEmpty
         ifFalse: [fn := fn first.
            Transcript show: "Reading sound from: ", dir, fn; cr.
           StoredSound fromFile: dir, fn]]]

Siren sounds inspect
Siren initializeSoundDictionary
```

Loading Spectral Data Files

```
[   | dir fn |
    dir := Siren soundDir, "sharc/tuba/".
   Transcript cr.
   ((FileDirectory on: dir) fileNamesMatching: ("*.spect")) do:
     [ :fn |
      Transcript show: "Reading sound from: ", dir, fn; cr.
      Spectrum fromFile: dir, fn]]

Siren spectra inspect
Siren Siren initializeSpectrumDictionary
```

Your Basic Siren Demo Script

To use this demo script, read through this 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.)

Set-up

Test the MIDI and sound I/O drivers.

```
[SirenUtility open] 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"

['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
```

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) play] d
```

Create 64 random events with parameters in the given ranges, play it on the default output voice.

```
[(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"
  ) play] 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
```

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') edit.] d
```

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)) play] 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 play ] 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;
  play ] 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) play ] 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 play ] d
```

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

EventGenerators

A cluster is the simplest event generator.

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

Chord object can give you an event list.

```
[((Chord majorTetradOn: 'f4' inversion: 0) duration: 1.0) edit]
[((Chord majorTetradOn: 'f4' inversion: 1) duration: 1.0) play]
```

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

```
[((Roll length: 2000 rhythm: 50 note: 60) ampl: 80) play] 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) play "edit"] 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) play]
```

Mark Lentczner's bell peals ring the changes.

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

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
```

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]
```

Display *lots* of random poly-lines in a very large space.
```
[DisplayList polylineExampleHuge]
```

Show the result of the IndentedListLayoutManager
```
[DisplayListView colorClassListExample]
```

Music Notations

Open a sequence view on a random event list.
```
[TimeSequenceView randomExample] d
```

Try the picth-time layout
```
[PitchTimeView randomExample]
[PitchTimeView openOnEventList: (EventList scaleExampleFrom: 48 to: 84 in: 10000)]
```

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]
[(EventList scaleExampleFrom: 36 to: 48 in: 3000) edit]
```

Sound Views

Create and view some example sounds
```
[SoundView openOn: SampledSound sawtooth]
[(SampledSound sweepDur: 10.0 rate: 44100 from: 10 to: 400 chans: 1) edit]
```

Read in a sound from a file

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

Utilities

```
ANO
  [MIDIPort allNotesOff]

Close down and clean up.
  [MIDIPort cleanUp]
```

Building a Siren Image

To load Siren into a VisualWorks 7.X virtual image, follow these steps.

```
Start VW 7.2
  Load your favorite parcels (AT tools, DB, etc.)
  Siren requires DLLCC and the GHeeg Namespaces ComposedTextEditor
  Load Siren.pcl
   File in the MusicConstants.st file

By-hand init
   Soundfile initializeSoundFileFlags
   SirenUtility initialize (or SirenUtility initializeSirenSTP)

Load optional L&F hacks

Open list workbook
  ListWorkBook open
  Use the Page/load_all menu item to load the workbook contents
    from the BOSS file Workbook.bos

Set up a new changelist
make a snapshot...
```

Reference, Acknowledgments

The main Siren Web site is at http://www.create.ucsb.edu/Siren.
This outline is on-line at http://www.create.ucsb.edu/Siren/7.2/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.smalltalk.org/
- http://www.fja-odateam.com/links/smalltalk/s_tutori.html
- http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/smalltalk1.html
- http://daitanmarks.sourceforge.net/or/squeak/squeak_tutorial.html
- http://www.cincom.com/scripts/smalltalk.dll/tutorial/version7/tutorial1/home.htm
- http://www.cs.ucf.edu/courses/cop4331/hughes/Summer1998/Tutorials/Smalltalk/index.html
- http://www.phaidros.com/DIGITALIS

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
Curtis Roads
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 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@UCBerkeley (1993-95), CREATE@UCSantaBarbara (1996-present), and the Technical University of Berlin (2000).

Siren 7.2 Release Notes

Current Status

This is the V7.2 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, so that the event list and event generator examples run well.

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.

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
- 7.2--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--The Lagoon, Palo Alto, 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

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

SuperCollider: an efficitne synthesis and processing language very similar to Smalltalk
http://create.ucsb.edu/CSL

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

A Note on Reading the Smalltalk-80 Programming Language

(Version 0.1 - stp - 12/91 -- improvised, improvements graciously solicited)

Smalltalk-80 is a rather wierd object-oriented message-passing 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.

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. A Smalltalk- 80 application is a network of objects; all action is caused by objects sending messages to each other. 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]. 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.

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

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)].

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.

For more info, there are several excellent on-line Smallatlk tutorials (just ask Google), see the reference section of this workbook.

Known Bugs in Siren 7.2

Kernel Classes
Pretty stable

Support Classes
Stale DB code
Some unused code in SirenUtility

IO and Voices
MIDI file reader is broken
MIDI input is untested
PortAudioPort sound I/O isn't finished
Sound file output -- samples are shifted (1s vs 2s complement?)

Graphics and GUI
The GUI views are "demo-quality" -- most are partially functional (output only), and have bugs
Selection in editors not ported
Polyline display doesn't zoom
Double redisplay on zoom of pitch-time views

List of ThingsToDo
Port IO drivers to other platforms
Port FFTW and spectrum classes
Implement polymorphic interpolating math for Functions
Port mixer classes
Complete the EventListTreeEditor
Add Siren icon to launcher toolbar