

Music and Sound Processing in Squeak Using Siren

Stephen Travis Pope

Center for Research in Electronic Art Technology (CREATE)

University of California, Santa Barbara

Introduction

The Siren system is a general-purpose music composition and production framework integrated with Squeak Smalltalk (1); it is a Smalltalk class library of about 320 classes (about 5000 methods) for building various music- and sound-related applications. Siren can be used on all Squeak platforms with or without support for MIDI or audio input/output (I/O). The system's Smalltalk source code is available for free on the Internet; see the Siren package home page at the URL <http://www.create.ucsb.edu/Siren>.

This chapter is divided into several sections: (a) the Smoke music description language used within Siren, (b) Siren's real-time MIDI and sound I/O facilities, (c) the graphical user interfaces (GUIs) for Siren objects, and (d) Siren applications—music/sound databases, and how a composer might use Siren. The presentation is intended for a Squeak programmer who is interested in music and sound applications, or for a computer music enthusiast who is interested in Squeak. Many references are provided to the literature on music software and object-oriented programming

Why Siren?

The motivation for the development of Siren is to build a powerful, flexible, and portable computer-based composer's tool and instrument. In the past, real-time musical instruments such as acoustic instruments or electroacoustic synthesizers have had quite different characteristics from software-based research and production tools. (There is of course an interesting gray area, and also a basic dependency on how the implement is applied—hammer-as-instrument, or piano-as-tool.) One possible description of the basic differences between tools and instruments is given in ~~the table below~~ Table 1.

Characteristic	Tool	Instrument
Typical application	construction	expression
Range of application	(hopefully) broad	narrow is acceptable
User interface	simple, standard	potentially complex but learnable
Customizability	none or little	per-user customizability
Application mode	for planned tasks	exploratory, experimentation

Table 1: Differences between Tools and Instruments

The Siren system is designed to support composition, off-line (i.e., non-interactive) realization, and live performance of electroacoustic music with abstract notations and musical structure representation, as well as sound and score processing. Other desired application areas include mu-

Music and Sound Processing Using Siren

sic/sound databases, music analysis, and music scholarship and pedagogy. It should ideally be flexible and formal like a tool, and support expression and customizability like an instrument.

The technical goal of the software is to exhibit good object-oriented design principles and elegant state-of-the-art software engineering practice. It needs to be an easily extensible framework for many kinds of intricately structured multimedia data objects, and to provide abstract models of high-level musical constructs, and flexible management of very large data sets.

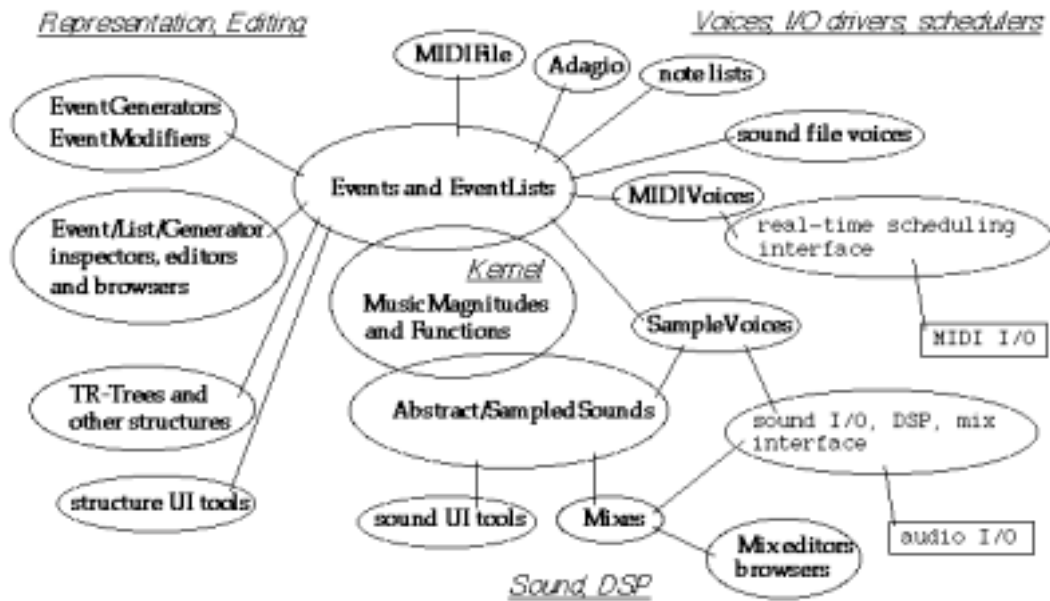
Elements of Siren

There are several packages that make up the Siren system:

- a general-purpose music representation system—the Smoke music representation language (the name, suggested by Danny Oppenheim, is derived from “Smallmusic object kernel”), which consists of music magnitudes, events, event lists, generators, functions, and sounds;
- a collection of I/O-related objects such as voices, schedulers, and drivers—real-time and file-based I/O objects for sound and MIDI;
- GUI components for musical applications—an extended GUI framework, widgets, and tools; and
- a collection of built-in end-user application prototypes—Squeak editors and browsers for Siren objects.

The Figure below-1 shows the basic components of Siren. At the center are the class categories of the Smoke music and multimedia representation language. On the left side are the editors and applications for manipulating Smoke objects in Siren. To the right are the various objects that handle input-output in various formats and real-time drivers. The components on the lower-right that are shown in courier typeface are low-level I/O driver interfaces written in C.

 Music and Sound Processing Using Siren


 Figure 1: Siren Architecture

 History and Relation to Composition

Siren and its direct hardware and software predecessors stem from music systems that developed in the process of my composition. Starting in the mid-1970s, I used non-real-time software sound synthesis programming languages for my composition; I had the very good luck to learn the Music10 language that was based on SAIL (Stanford AI Language) and ran on DEC PDP-10 mainframe computers. This was a very advanced (for its time) and flexible programming language that served as a synthesis and a **score** score-generation language at the same time. I quickly became accustomed to building special representations for each composition and to having flexible high-level tools for music. When I moved onto DEC PDP-11-series mini-computers running UNIX in the late 1970s, I was forced to start writing my own composition tools in the highest-level languages to which I had access. (This led to having to do my own ports of Lisp and Smalltalk virtual machines, but that's another chapter altogether.) The ramification of this is that the software I describe below (including Siren) is always oriented primarily towards the representation of musical structures and compositional algorithms, with real-time performance or actual sound synthesis as secondary considerations. Siren today remains primarily a music description language and composition structuring tool, with a few MIDI I/O features, simple graphical tools, and hooks to other Squeak facilities for sound generation.

My first generation of object-oriented music software, called ARA, was an outgrowth of a Lisp rule-based expert system I wrote between 1980 and 1983 for the composition of *Bat out of Hell* (2). ARA had a rather inflexible music representation (designed for a specific software instrument written in the Music11 sound synthesis language), but allowed the description and manipulation of “middle-level” musical structures such as chords, phrases, and rhythmical patterns in the rules of the expert system.

Music and Sound Processing Using Siren

The next generation was the DoubleTalk system (3, 4), which used a Smalltalk-80-based Petri net editing system built (in Berkeley Smalltalk) by Georg Heeg et al. at the University of Dortmund, Germany. I used DoubleTalk for the composition of *Requiem Aeternam Dona Eis* (1986). This system allowed me to “program” transitions between states of the compositional system using a Prolog-like declarative language that was used to annotate the Petri nets. To execute a DoubleTalk net, one defined its initial “marking”—the placement and types of tokens distributed among the nodes—and then ran the Petri net simulator, the result of which was a score generated by the simulated net’s transitions.

In 1986, I started working at Xerox PARC and also at the Stanford University Center for Computer Research in Music and Acoustics (CCRMA). I ~~also~~ wrote the first flexible version of a Smalltalk music description language while there, which served as the foundation for the HyperScore ToolKit (5). This package was used (among others) for the composition *Day* (1988), and it was the first to support real-time MIDI I/O as well as graphical notations. (This was the only composition for which I have used MIDI, and the last for which I tried to build graphical tools. I have only done the minimum to support MIDI and GUIs in my tools since 1988.)

Siren’s direct predecessor, known as the MODE (Musical Object Development Environment) (6, 7), was used for *Kombination XI* (1990/98) (8) and *All Gates Are Open* (1993/95). The MODE was based on ParcPlace Systems’ VisualWorks implementation of Smalltalk and supported sound and MIDI I/O as well as specialized interfaces (via user primitives) to sound analysis/resynthesis packages such as a phase vocoder, and to special-purpose mixing hardware such as the Studer/Dyaxis MacMix system.

In each of these cases, some amount of effort was spent—after the completion of a specific composition—to make the tools more general purpose, often making them less useful for any particular task. Siren (9, 10) is a re-implementation of the MODE undertaken in 1997-9; it is based on the representations and tools ~~Im~~I am using in the realization of *Ywe Ye, Yi Jr Di* (work in progress). The “clean-up” effort was minimized here; the new Siren package is much more useful, but for a much smaller set of tasks and attitudes about what music representation and composition are. If Siren works well for other composers, it is because of its idiosyncratic approach, rather than its attempted generality (i.e., the instrument approach, rather than the tool approach).

Siren and its predecessors are documented in the book “The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology” (11), in a series of papers in the Proceedings of the 1982, 1986, 1987, 1989, 1991, 1992, 1994, 1996, 1997, and 1999 International Computer Music Conferences (ICMCs), in an extended article in *Computer Music Journal* from 1992 (6), and in the 1997 book *Musical Signal Processing* (12). Many of these papers ~~of these~~, and related documents are available from the Web URL <http://www.create.ucsb.edu/~stp/publ.html>.

Programming Languages for Music

In the computer music literature (e.g., 11), the primary programming languages used for advanced experimental systems (to this day) have been Lisp

Music and Sound Processing Using Siren

and Smalltalk; this can be traced to several basic concepts. Both languages provide an extremely simple, single-paradigm programming model (i.e., all data are of the same basic “type” and all behavior is accessed in the same way), and both have consistent syntax that scales well to large expressions (this is a matter of debate among “language bigots”). Both can be interpreted or compiled with ease and are often implemented within development environments based on one or more interactive “read-eval-print loop” objects. The history of the various Lisp machines and Smalltalk-based operating systems demonstrates the scalability of Lisp and Smalltalk both “up” and “down,” so that everything from high-level applications frameworks to device drivers can be developed in a single language system. The Smalltalk history shows the independent development of the programming language, the basic class libraries, the user interface framework, and the delivery platform across at least four full generations.

Two important language features that are common to both Lisp and Smalltalk are dynamic typing and dynamic polymorphism. Dynamic typing means that data type information is specific to (run-time) values and not to (compile-time) variables as in many other languages. In Pascal or C, for example, one declares all variables as typed (e.g., `int i;` means that variable `i` is an integer) and may not generally assign other kinds of data to a variable after its declaration (e.g., `i = “hello”;` to assign a string to `i`). Declaring a variable name in Lisp or Smalltalk says nothing about the types of values that may be assigned to that variable. While this generally implies some additional run-time overhead, dynamic binding is a valuable language asset because of the increase it brings in software flexibility, abstraction, and reusability.

Polymorphism means being able to use the same function name with different types of arguments to evoke different behaviors. Most standard programming languages allow for some polymorphism in the form of overloading of their arithmetical operators, meaning that one can say `(3 + 4)` or `(3.1 + 4.1)` ~~in order~~ to add integers or floating-point numbers. The problem with limited overloading (limited polymorphism) is that one is forced to have many names for the same function applied to different argument types (e.g., function names like `playEvent()`, `playEventList()`, `playSound()`, `playMix()`, etc.). In Lisp and Smalltalk (as well as several other ~~object~~ object-oriented languages) all functions can be overloaded, so that one can create many types of objects that can be used interchangeably (e.g., many different types of objects can handle the `play` message in their own ways). Using polymorphism also incurs a run-time overhead, but, as with dynamic binding, it can be considered essential for a language ~~on which to~~ that will be used as the base for an exploratory programming environment for music and multimedia applications.

The 1991 book "The Well-Tempered Object" (11) describes the second generation of O-O music software systems (mid- to late-1980s, the first generation having started in the mid-1970s), and there are several third-generation systems in both LISP (Stanford's CLM or CMU's Nyquist) and Smalltalk (the Symbolic Sound Kyma system, Dmix, Siren).

Representation of Multimedia Objects

There is a rich and diverse literature related to the representation, manipulation, and interchange of multimedia data in general, and musical sound and

Music and Sound Processing Using Siren

scores in particular. Two useful surveys are those by Roger Dannenberg (13) and Geraint Wiggins et al. (14).

Among the important issues are (a) which media related to sound and music are to be supported—recorded sound, musical performance, musical structure; (b) what level of sonic semantics and musical structure is to be supported, and (c) how exactly the representation is to capture an actual performance or recording. In many systems, issue (a) is addressed by a small and fixed set of data types (e.g., sound-only, control-only, or event-only), and a trade-off is seen between issues (b) and (c)—, that is, between what Wiggins et al. call “structural generality” and “representational completeness.”

Many object models for complex domains (musical or not) start by defining a set of classes of objects that represent the basic “units of measure” or magnitudes of the domain. In the case of sound and music, this means classes to model the basic properties of sounds and musical events such as time (and/or duration), pitch, loudness, and spatial dimensions. Along with some model of generic “events” (at the level of words or musical notes), one must build micro-level functions and control objects, and higher-level “event lists” to represent sentences, melodies, and other composite events. This basic event/event-list design is very similar to the design patterns found in graphics systems based on display lists.

The Smoke Music Representation Language

The “kernel” of Siren is the set of classes for music magnitudes, functions and sounds, events, event lists, and event structures known as the Small-music object kernel, or (~~Smoke—the name was suggested by Danny Oppenheim~~) (15). Smoke is described in terms of two related description languages (verbose and terse music input languages), a compact binary interchange format, and a mapping onto concrete data structures. All of the high-level packages of Siren—event lists, voices, sound/DSP, compositional structures, and the user interface framework—interoperate using Smoke events and event lists.

Smoke supports the following kinds of description:

- abstract models of the basic musical quantities (scalar magnitudes such as duration, pitch, loudness or duration);
- instrument/note (voice/event or performer/score) pattern for mapping abstract event properties onto concrete parameters of output media or synthesis methods;
- functions of time, sampled sound, granular description, or other (non-note-oriented) description abstractions;
- flexible grain-size of “events” in terms of “notes,” “grains,” “elements,” or “textures”;
- event, control, and sampled sound description levels;
- nested/hierarchical event-tree structures for flexible description of “parts,” “tracks,” or other parallel/sequential organizations;
- separation of “data” from “interpretation” (what vs. how in terms of providing for interpretation objects called voices);

Music and Sound Processing Using Siren

- abstractions for the description of “middle-level” musical structures (e.g., chords, clusters, or trills);
- annotation of event tree structures supporting the creation of heterarchies (lattices) and hypermedia networks;
- annotation including graphical data necessary for common-practice notation; and
- description of sampled sound synthesis and processing models such as sound file mixing or DSP.

Given a flexible and abstract basic object model for Smoke, it should be easy to build converters for many common formats, such as MIDI data, formatted note lists for software sound synthesis languages (16), DSP code, or mixing scripts. Additionally, it should be possible to parse live performance data (e.g., incoming MIDI streams) into Smoke objects, and to interpret or play Siren objects (in some rendition) in real-time.

The “executive summary” of Smoke from (15) 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; 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. Voice objects and applications determine the interpretation of events’ properties and may use “standard” property names such as pitch, loudness, voice, duration, or position.

Events are grouped into event collections or event lists 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, etc.); 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. Voices map event properties onto parameters of I/O devices; there can be a rich hierarchy of them. A scheduler expands and/or maps event lists and sends their events to their voices.

Sampled sounds are also describable, by means of synthesis “patches,” or signal processing scripts involving a vocabulary of sound manipulation messages.

Smoke objects also have behaviors for managing several special types of links, which are seen simply as properties where the property name is a symbol such as **usedToBe**, **isTonalAnswerTo**, or **obeysRubato**, and the property value is another Smoke object, for example, an event list. With this facility, one can build multimedia hypermedia navigators for arbitrary Smoke networks. The three example link names shown above could be used to implement event lists with version history, to embed analytical information in scores, or to attach real-time performance controllers to event lists, respectively.

Music Magnitudes

MusicMagnitude objects are characterized by their identity, class, species, and value. For example, the pitch object that represents the note named c3 has its particular object identity, is a member of class **SymbolicPitch**, of the species **Pitch**, and has 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.* that is, 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—for example, (**440.0 Hz asMIDIKeyNumber**)—and mixed-mode operations—for example, (**1/4 beat + 80 msec**)—are defined.

The basic abstract model classes include **Pitch**, **Loudness**, and **Duration**. These classes are abstract and *don't* do not even have subclasses; they signify what kind of property is being represented. They are used as the species for families of classes that have their own inheritance hierarchy based on how they represent their values. This framework is easily extensible for composition- or notation-specific magnitudes.

The Figure below-2 shows the abstract “representation” class hierarchy on the left and the concrete “implementation” hierarchy on the right. The lines between the two sides denote the species relationships, *e.g.* for example, both **HertzPitch** and **SymbolicPitch** are of species **Pitch**, and can defer to the species for performing mixed-mode operations such as (**#c4 pitch + 12 Hz**). The representation hierarchy has abstract classes such as **Chroma** (species classes representing objects for pitch and temperament), while the implementation hierarchy has abstract classes such as **Interval-Magnitude**, which generalizes the concrete classes with fixed numerical ranges.

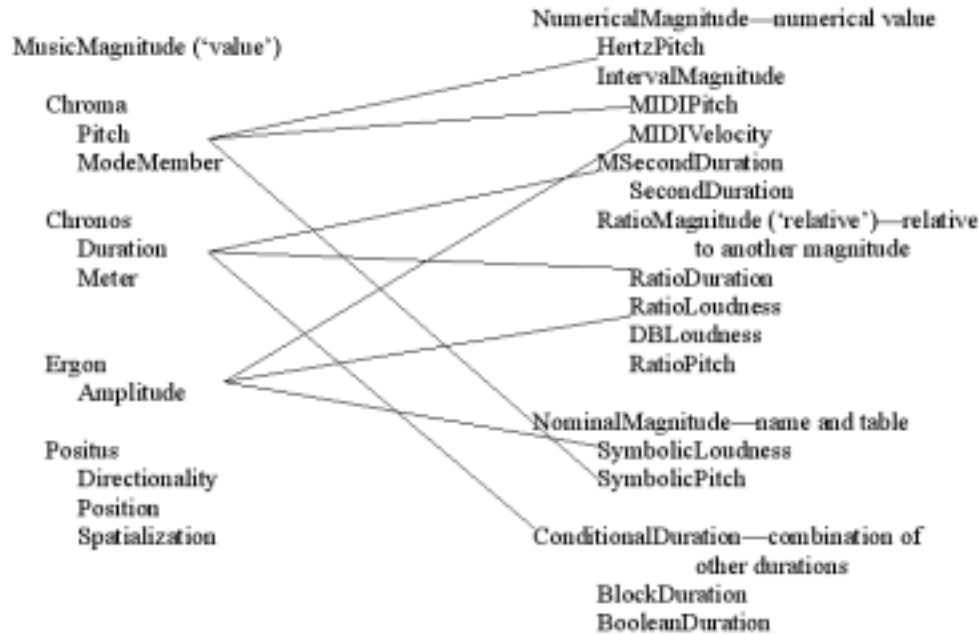
 Music and Sound Processing Using Siren


Figure 2: Class-Species and Subclass Inheritance Relationships among Siren MusicMagnitudes

The handling of time as a parameter is finessed via the abstraction of duration. All times are represented as durations of events, or delays between events, 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 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 **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 subsequent examples.

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

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.”

Mixed-mode Arithmetic

Music and Sound Processing Using Siren

(1/2 beat) + 100 msec	“ (0.6 beat)”
'a4' pitch + 25 Hz	“ (465.0 Hz)”
(#a4 pitch + 100 Hz) asMIDI	“ (73 key)”
'mp' ampl + 3 dB	“ (-4.6 dB)”

Events and Event Lists

The **AbstractEvent** 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 primary messages that events understand are: **(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). This kind of “generic behavior” is implemented by overriding the method **doesNotUnderstand**, which is both very useful and rather dangerous (as it can make broken methods quite difficult to debug).

The meaning of an event’s properties is interpreted by voices and user interface objects; it is obvious that, for example, ~~(e.g.,)~~ a pitch property 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 “;” 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. (The message “=>” is similar to the standard Smalltalk “->” message except that a special kind of Association [an EventAssociation] is created.) This kind of convenient and compact expression is simply Smalltalk syntax using a few additional implementors of the message “;” for concatenation, as shown in ~~the~~ Table 2 below.

Receiver class	Answer	Example
MusicMagnitude	MusicEvent	440 Hz, (1/4 beat), 44 dB
MusicEvent	EventList	event1, event2
EventAssociation	EventList	(dur1 => evt1), (dur2 => evt2)

Table 2: Interpretations of Concatenation Messages in Smoke

Music and Sound Processing Using Siren

The classes Siren uses for events are as follows.

AbstractEvent — Object with a property list (lazily created)

DurationEvent — adds duration instance variable

MusicEvent — adds pitch and voice instance variables

ActionEvent — has a block that it evaluates when scheduled

It is seldom necessary to extend the hierarchy of events. Examples of verbose and terse Siren event creation messages are given below.

Verbose Event Creation Messages — Class **messages** 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** — Concatenation of ~~music~~ **Magnitudes**

440 Hz, (1/4 beat), 44 dB.

"Simple event"

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

(#embrochure -> #tight).

"with an added (arbitrary) property"

(#c4 pitch, 0.21 sec, 64 velocity)

voice: Voice default.

"Event using different syntax"

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 event list and that of the event). The event list classes are subclasses of **DurationEvent** themselves. This means that event lists can behave like events and can therefore be arbitrarily deeply nested, that is, 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

Music and Sound Processing Using Siren

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 by Carla Scaletti in [17]). 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” as suggested by Roger Dannenberg (13).

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 triad (i.e., create a named event list and add three events that all start 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
```

This same chord (this time anonymous) could be defined more tersely using simple concatenation of event associations (note the comma between the associations),

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

This chord could have been created 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) associations looks like this:

```
⌞
(440 Hz, (1/2 beat), 44.7 dB), “comma between events”
(1 => ((1.396 sec, 0.714 ampl) phoneme: #xu))
```

EventGenerators and EventModifiers

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, or osti-

Music and Sound Processing Using Siren

nati. Each event generator subclass knows how it is described—for example, 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/modifiers are described in more detail in (18); some of the other issues are discussed in (19).

EventGenerator Examples

Chords are simple one-dimensional event generators.

```
((Chord majorTetradOn: 'f4' inversion: 1) duration: 1.0) play
```

Play a drum roll—20 beats/sec (50 msec each) for 2 sec, middle-C, loud

```
((Roll length: 2 sec rhythm: 50 msec note: 60 key) ampl: #ff) play
```

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

Given interval ranges for selection of pitch and amplitude; play a constant rhythm.

```
(Cloud dur: 6 "Cloud lasts 6 sec."
  pitch: (48 to: 60) "with pitches selected from this range"
  ampl: (80 to: 120) "and amplitudes in this range"
  voice: (1 to: 8) "select from these voices"
  density: 5) "play 5 notes per sec."
```

Play a 6-second cloud that goes from low to high and soft to loud—give starting and ending selection ranges for the properties.

```
(DynamicCloud dur: 6 "6-second DynamicCloud generator"
  pitch: #((30 to: 44) (60 to: 60)) "with starting and ending pitch"
  ampl: #((20 to: 40) (90 to: 120)) "and amplitude ranges"
  voice: (1 to: 4) "single interval of voices"
  density: 15) play "play 15 per second"
```

Edit a dynamic selection cloud that makes a transition from one triad to another—give starting and ending pitch sets for the selection.

The result is displayed in Figure 3(a) in a piano-roll-like notation editor.

```
(DynamicSelectionCloud dur: 4 "6-second DynamicSelectionCloud"
  pitch: #(#(48 50 52) #(72 74 76)) "give sets for pitch selection"
  ampl: #(60 80 120) "constant selection set"
  voice: #(1 2) "2 MIDI voices"
  density: 20) edit "20 notes per sec"
```

Rubato example: apply a “slow-down” tempo-map to a drum roll by scaling the inter-event delays

```
| roll rub | "Create a drum roll with 10 beats/sec"
roll := ((Roll length: 2000 rhythm: 100 note: 60) ampl: 80) eventList.
"Create a modifier to slow down by a factor of 1.5"
rub := Rubato new function: (LinearFunction from: #(0 1) (1 1.5)); scale: 10.
rub applyTo: roll. "Apply the event modifier to the event list."
```

 Music and Sound Processing Using Siren

roll explore

"Explore the result"

The object explorer shown in Figure 3(b) illustrates the inner structure of a Siren event list; the main instance variables are at the top of the property list, and one of the list's event is expanded to show its internal state. The effect of the decelerando function is visible as the increase of the differences between the relative start times of the events in the list.

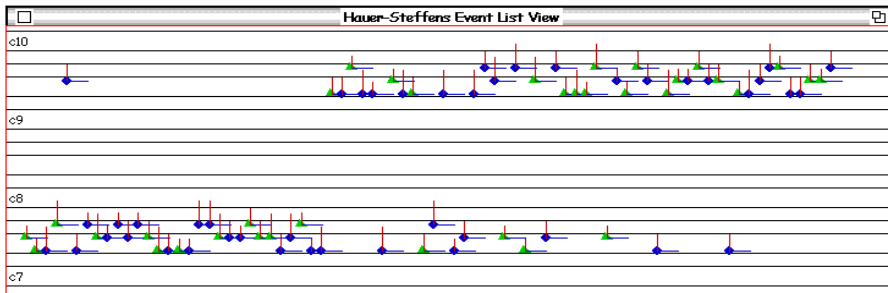


Figure 3: (a) DynamicSelectionCloud Chord Cross-fade

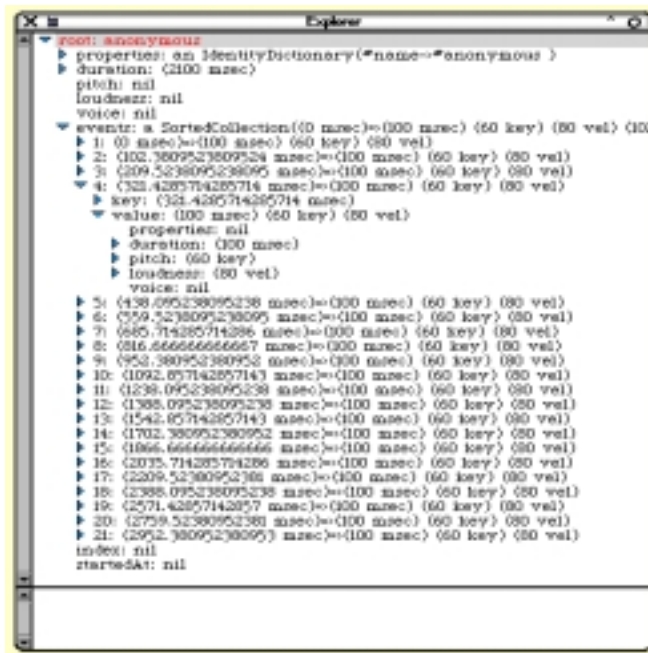


Figure 3: (b) EventList Result from Applying a Decelerando Tempo Map to a Drum Roll

 Siren I/O

The “performance” of events takes place via **IOVoice** 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 score file storage for-

Music and Sound Processing Using Siren

mats—such as note lists for various software sound synthesis languages (16), MIDI file format, or phase vocoder scripts—or for use with real-time schedulers connected to MIDI or sampled sound 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. **IOVoice** objects can also read input streams (e.g., real-time controller data, or output from a co-process) 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.

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` (written entirely in Squeak) does just this; it can be used to sequence and synchronize event lists that may include a variety of voices.

IOVoice Examples

Create a random event list and write it out to a cmix-format notelist file.

```
| file list voice |
file := FileStream named: 'test.out.cmix'.
    "This example creates an event list with random notes."
list := EventList randomExample: 64.
    "Create a voice to map the list to a cmix-format score file."
voice := CmixVoice newNamed: '1' onStream: file.
    "Store the event list on the voice's file."
voice play: list.
file close.    "Close the file."
```

The resulting cmix score file looks like,

```
/* cmix MINC data file created 8 May 2000 */
system("sfcreate -r 44100 -c 1 -i out.snd");
output("out.snd");          /* output sound file */
makegen(1, 10, 1024, 1)     /* f1 = sine wave */

ins(t, 0.0, 0.264, 79, 0.653401229834836, 0.4343151734556608, 13039);
ins(t, 0.264, 0.255, 74, 0.2897873042569436, 0.39283562343234, 22197);
ins(t, 0.519, 0.281, 75, 0.4070028436402803, 0.399486889950692, 18610);
ins(t, 0.8, 0.232, 77, 0.9441084940657525, 0.421033562096317, 22386);
ins(t, 1.032, 0.28, 73, 0.815713333345816, 0.431038966901153, 22444);
ins(t, 1.312, 0.298, 70, 0.900661926670308, 0.3880002476591618, 12011);
ins(t, 1.61, 0.248, 72, 0.05989623258816834, 0.4281569774952516, 14359);
... etc.
```

Real-time music I/O in Siren is managed by low-level interfaces to the host operating system’s device drivers for sound and MIDI; **Siren** objects use primitive methods that call out to the external functions. The glue code for these primitives is written in Smalltalk and translated to C for linking with the Squeak virtual machine (itself written in Smalltalk and translated). Several sets of primitives exist for Squeak on various platforms, including

Music and Sound Processing Using Siren

support for sound synthesis, digital audio signal processing, MIDI event-oriented and continuous controller I/O, and VM-level scheduling.

There are rich class libraries for managing MIDI I/O connections, as shown in the code example below, which creates a port, a device, and a voice in order to play an event list. Object models for MIDI controllers, extended MIDI commands, and GeneralMIDI channel maps are provided in Siren as well, as demonstrated in the following code fragments.

MIDI voice Example

```
| voice device port scale |
port := MIDIPort default.      "Create a MIDI port"
device := MIDIDevice on: port.  "Create a MIDI device"
voice := MIDIVoice on: device.  "Create a MIDI voice"
                                "Create an example event list, a scale"
scale := EventList scaleExampleFrom: 24 to: 60 in: 2000.
                                "Play the event list on the voice"

voice play: scale
```

Method MIDIPort>>testBend

```
"Demonstrate pitch-bend by playing two notes and bending them."
"MIDIPort testBend"
```

```
| port start |
port := self default.
port open.
```

```
"Set the recorder MIDI instrument."
```

```
port programChange: 0 to: 'Recorder'.
port programChange: 1 to: 'Recorder'.
start := Time millisecondClockValue + 150.
```

```
"Play two notes."
```

```
port play: 76 at: start dur: 5000 amp: 60 voice: 0.
port play: 80 at: start dur: 5000 amp: 60 voice: 1.
```

```
"Bend them--one up, one down."
```

```
0 to: 500 do:
  [:i |
    port pitchBend: 0 to: 4096 + (i * 8) at: nil.
    port pitchBend: 1 to: 4096 - (i * 8) at: nil.
    (Delay forMilliseconds: 10) wait]
```

Example of GeneralMIDI instrument banks: load channels 1-16 with tuned percussion instruments

```
MIDIPort setEnsembleInOrder:
```

```
#(Agogo 'Tinkle Bell' Timpani Xylophone
  Applause 'Taiko Drum' Glockenspiel 'Synth Drum'
```

Music and Sound Processing Using Siren

Gunshot 'Steel Drums' Helicopter Vibraphone
 Woodblock 'Telephone Ring' Kalimba 'Blown Bottle')

Applications can have direct access to the Siren real-time I/O scheduler, for example, to add an event list at a specific future time and kick-start the scheduler, as in the following example.

```
Siren schedule "Get the 'global' real-time scheduler."
              "Add an example event list at some future time."
addAppointment: ActionEvent listExample
in: (1000 msec);
              "start the schedule in case it's off."
runAppointments
```

Because events are medium independent, and voices manage all the details of output channels, we can write the “multimedia” example below. The goal here is to generate and play a mixed-voice event list; a cloud event generator plays alternating notes on a MIDI voice and via the built-in sound synthesis primitives, and a parallel list of action events flashes random screen rectangles in parallel with the sound and MIDI output.

```
|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"
      "(int 1 means MIDI channel 1).")
voice: (Array with: 1 with: (SynthVoice default))
density: 5) eventList.     "Play 5 notes/sec. and get the events."
                          "Add some 'action' events, this example's
                          events draw ractangles on the screen"
el addAll: ActionEvent listExample2.
el play                    "and play the merged event list"
```

User Interfaces for Music/Sound Processing

Navigator MVC in Siren

The Smalltalk-80 Model-View-Controller (MVC) user interface paradigm (20) is well known and widely imitated. The traditional three-part MVC architecture involves a model object representing the state and behavior of the domain model—in our case, an event list or signal. The view object presents the state of the model on the display, and the controller object sends messages to the model and/or the view in response to user input.

Many Smalltalk applications extend this design pattern to use a separate object to model the GUI and selection state for the model (giving us four-part MVC); these manager objects are often referred to as browsers, inspectors, or editors.

Music and Sound Processing Using Siren

“Navigator MVC” (21) (see [the Figure 4 below](#)) is a factoring of the controller/editor and view for higher levels of reuse. The traditional MVC components are still there, and are connected by the smalltalk dependency mechanism (shown in gray). With this architecture and design pattern for MVC, most applications are modelled as enhanced display list editors (i.e., the generic tool is “smart draw”), with special layout manager objects for translating the model structure into a graphical display list representation and for translating structure interaction into model manipulation.

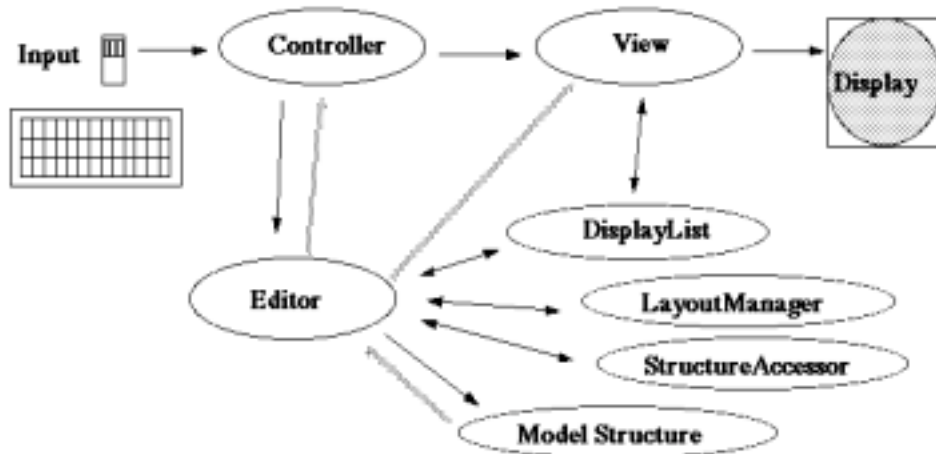


Figure 4: Navigator MVC Architecture

A **StructureAccessor** is an object that acts as a translator or protocol converter. An example might be an object that responds to the typical messages of a tree node or member of a hierarchy (e.g., What’s your name? Do you have and children/sub-nodes? Who are they? Add this child to them.). One specific, concrete subclass of this might know how to apply that language to navigate through a hierarchical event list (by querying the event list’s hierarchy).

The role of the **LayoutManager** object is central to building Navigator MVC applications. Siren’s layout manager objects can take data structures (like event lists) and create display lists for time-sequential (i.e., time running left-to-right or top-to-bottom), hierarchical (i.e., indented list or tree-like), network or graph (e.g., transition diagram), or other layout formats. The editor role of Navigator MVC is played by a smaller number of very generic (and therefore reusable) objects such as **EventListEditor** or **SampledSoundEditor**, which are shared by most of the applications in the system.

Much of the work of building a new tool within the Siren system often goes into customizing the interaction and manipulation mechanisms, rather than just the layout of standard pluggable view components. Building a new notation by customizing a layout manager class and (optionally) a view and controller, is relatively easy. Adding new structure accessors to present new perspectives of structures based on properties or link types can be used to extend the range of applications and to construct new hypermedia link navigators. This architecture means that views and controllers are extremely generic (applications are modeled as structured graphics editors), and that the bulk of many applications’ special functionality resides in a small number of changes to existing accessor and layout manager classes.

Siren MVC Examples

The example screens below (Figure 5) show the simple Siren display list editor running under Squeak MVC; it allows you to manipulate hierarchical structured graphics objects. The pop-up menu in the right of the view shows the default display list controller message. Keyboard commands and mouse interaction support zooming and scrolling. One item is selected in the view, and can be dragged or resized using its “selection handles.”



Figure 5: Siren DisplayListView Example

The example below in Figure 6 shows a class inheritance hierarchy presented as a left-bound tree. Color is used to denote class species relationships in the class hierarchies; this is determined by the layout manager used for this example. A refined tree layout manager could do graphical balancing or top-down layout.



Figure 6: Siren LayoutManager Example

A time sequence view is a display list view ~~that~~ whose layout manager interprets time as running from left to right. In the example below (Figure 7), the time sequence is derived from the sentence "Und die Fragen sind die Sätze, die ich nicht aussprechen kann."

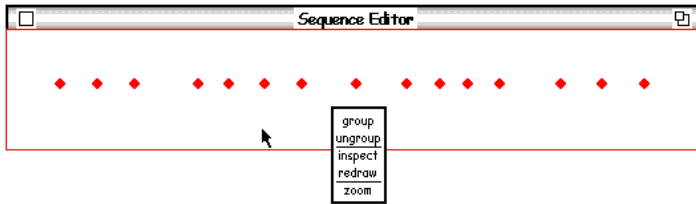


Figure 7: Siren TimeSequenceView Example

In a pitch/time view, time runs left-to-right, and pitch is displayed from bottom-to-top. In the example **below** in Figure 8, the layout manager creates a separate sub-display-list for each note, adding lines to the note head image to show its amplitude and duration, several other properties, and the amplitude envelope.

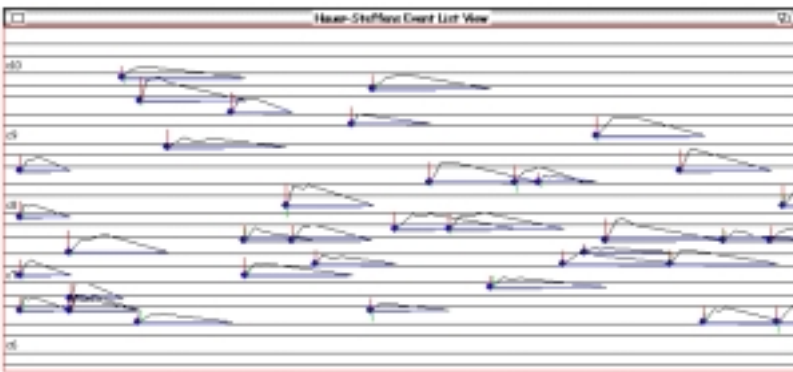


Figure 8: Siren Score Editor Example

The multi-function view allows the viewing and editing of up to 4 functions of 1 variable. The example (Figure 9) shows linear break-point functions in red and yellow, an exponential segment function in blue, and a cubic spline function in green. The buttons along the left are for selecting a particular function for editing or file I/O.

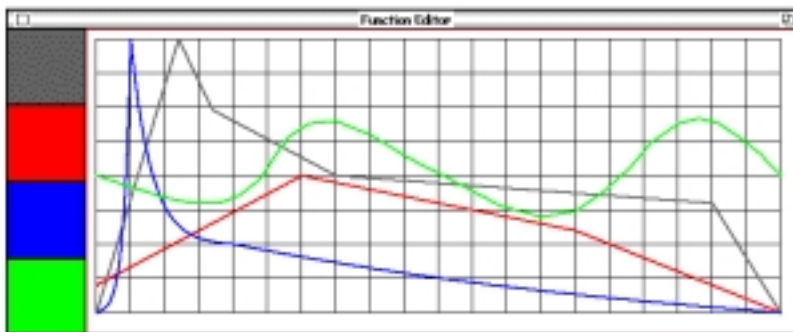


Figure 9: Siren Function Editor Example

The sonogram view displays an FFT-derived spectrum. **In the example below** Figure 10 shows the spectrum of a swept sine wave.

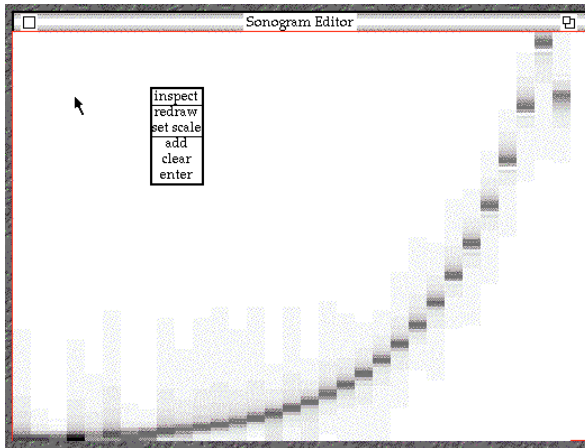


Figure 10: Siren Sonogram Example

Examples of Siren Applications

In this section, we describe two application areas for Siren: (a) a music/sound database project that uses Siren together with other languages and external interfaces, and (b) an application of Siren for composition.

Sound/Music Databases: Paleo

Most prior work in sound or music databases has addressed a single kind of data (e.g., MIDI scores or sampled sound effects) and has pre-defined the types of queries that are to be supported (e.g., queries on fixed sound properties or musical features). Earlier systems also tended to address the needs of music librarians and musicologists, rather than composers and performers. In the Paleo system under development since 1996, we have built a suite of sound and music analysis tools that is integrated with an object-oriented persistency mechanism in Squeak.

The central architectural feature of Paleo is its use of dynamic feature vectors and on-demand indexing. This means that annotational information derived from data analysis can be added to items in the database at any time, and that users can develop new analysis or querying techniques and then have them applied to the database's contents on-the-fly within a query. For data that is assumed to be musical sound, this might mean performing envelope detection, spectral analysis, linear prediction, physical model parameter estimation, transient modeling, etc and so on. For musical performance data (e.g., MIDI), this might entail extraction of expressive timing, phrase analysis, or harmonic analysis.

Paleo content is assumed to be sampled sound, musical scores, or captured musical performances. Scores and performance formats can be simple (e.g., MIDI-derived) or may contain complex annotation and embedded analysis. Paleo is specifically constructed to support multiple sets of captured musical performances (for use in comparing performance expression). This includes the derivation of basic timing and dynamics information from MIDI performances (to be able to separate the performance from the "deadpan" score), and the analysis of timbral information from recorded sounds.

Music and Sound Processing Using Siren

For score analysis, we use a variety of methods, including simple statistical models, rule-based analysis, and constraint derivation. Sampled sound analysis is undertaken using a suite of functions called NOLib that ~~are is~~ written in the MatLab language and can be accessed from within the Paleo environment over the net via socket-based MatLab servers. The techniques available in NOLib include all standard time-, frequency-, wavelet modulus-domain analysis operations, as well as pitch detection, instrument classification, and sound segmentation.

The two main applications we envision for Paleo are its use as an integrated data support system for composers, and in a performer's rehearsal workstation. The first set of applications will put the database at the core of a composition development environment that includes tools for thematic and sonic experimentation and sketch data management. The second platform centers on manipulating rehearsal performance data relative to a "reference" score (which may or may not be a "dead-pan" interpretation). Users can play into the system and then compare their performance to another one of their own or of their teacher's. Query preparation takes place using pre-built tools such as the composer's sketch browser, or by creating direct queries in a simplified declarative query language.

The implementation of the Paleo database persistency and access component is based on the public domain Minnestore object-oriented database package (22), which allows flexible management of data and indices. The Squeak port of Minnestore is called SMS (Squeak Minnestore).

Paleo applications can communicate with an SMS database server over a network and can pass sound sample data or event streams to ~~or~~ from the database. We currently use a simple socket-based protocol but plan to move to a CORBA-based distribution infrastructure in the near future.

To stress-test Paleo's analysis and query tools against a realistic-sized data set, the test contents included over 1000 scores of keyboard music (Scarlatti, Bach, Bartok, the Methodist hymnal, etc.), several hundred "world" rhythms, the SHARC database of instrument tone analyses, 100 recorded guitar performance techniques, flute performances, and spoken poetry in five languages. Most of the content is freely available on the Internet.

Paleo Architecture

In Paleo, as in Siren, music and sound data are represented via Smoke objects. In Paleo's SMS data persistency layer, Smoke objects are stored in object sets, which are akin to database tables. Each object set stores one kind of objects, and can have any number of stored or derived indices. The collection of all defined indices determines the feature vector of the object set. When stored to disk, each object set has its own directory, storage policy, and a group of index files. For performance reasons, there are also cache policies per object set, and methods exist for keeping active object sets in a RAM disk.

Various services can be used by the SMS database server, such as calls to the NOLib functions (see below) or the use of extra Smalltalk processes for data analysis. The SMS server really only provides the persistency layer and cache policies for open object sets. The overall architecture is as shown in Figure 11 ~~below~~.

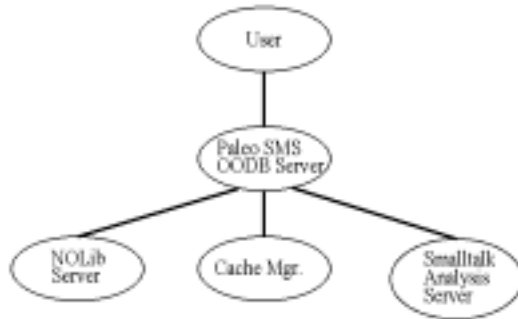


Figure 11: Paleo Architecture

The NOLib Analysis Functions

NOLib is a suite of data analysis and feature extraction routines written by Nicola Orio at CREATE in the MatLab programming language. These functions can be called by analysis scripts (interpreted MatLab programs), which can themselves be started by a network-based “analysis server.” We use the public-domain “octave” implementation of MatLab running on UNIX servers. MatLab was chosen for these analysis functions because of the excellent support for signal processing, simple and flexible file I/O, and portability of public-domain MatLab implementations. NOLib functions support sound analysis, recording segmentation, and instrumental timbre feature extraction (e.g., for analyzing the performance technique of instrumental performers).

MIDI File Analysis with Constraints

The purpose of the score analysis framework built for Paleo by Pierre Roy during 1999 is to allow complex queries on various kinds of musical data, including scores, in the spirit of the Humdrum system (23). A large amount of digitized music is available as MIDI files, for instance on any of the many MIDI archives on the Internet.

The MIDI format, however, provides only low-level musical information: it is **rather** a performance-oriented rather than an analysis-oriented representation of music. Thus, we need to analyze MIDI files **in-order** to compute additional musical features, such as: pitch-classes (by resolving enharmonic ambiguities), voice leading, keys, and harmonies.

The different tasks of analysis—enharmonic, melodic, tonal, and harmonic analysis—are not independent. For instance, the enharmonic analysis depends on tonal analysis, and conversely, the computation of local keys is based on the frequency of the different pitch-classes. Therefore, we need a global strategy in which the different tasks are performed simultaneously.

In the context of our analysis, we often need to perform only a partial analysis because many queries only involve a few specific elements or incomplete information. Consider the following queries: “How many of Scarlatti’s harpsichord sonatas end with a perfect cadence?” or “Are there more minor than major chords in the preludes of Bach’s *Well-Tempered Clavichord*?” In such cases, it is useless to perform a complete harmonic analysis of the 555 sonatas by Scarlatti, or of the 48 preludes of the *Well-*

Music and Sound Processing Using Siren

Tempered Clavichord. This mode of usage demands a scheme allowing partial and incomplete analysis.

What to analyze also depends on various parameters, such as the epoch, the style, and the nature (i.e., form, instrumentation) of the music being considered, for example, the anatomic limitations of human voice compared to a keyboard instrument. Our analysis strategy should be easily adaptable to various situations.

The previous remarks led us to an approach based on constraint satisfaction, instead of using specific algorithms for the different tasks of analysis (24). As a declarative paradigm, constraint satisfaction allows us to build systems that can be easily adapted to specific situations. For instance, adapting the system to vocal or keyboard music analysis is just a matter of using a different set of constraints for the melodies. Constraint resolution can also easily provide partial and incomplete analyses. More precisely, the query “How many sonatas by Scarlatti end with a perfect cadence?” will only require the computation of elements related to the last two chords of each sonata. Finally, constraint resolution is a global process, in which the different elements are progressively computed; thus, interdependent tasks are interlaced in the resolution.

A constraint satisfaction problem or CSP (25) consists of (a) a set of variables (each associated with a set of possible values—its domain), representing the unknown values of the problem, and (b) a set of constraints, expressing relationships between the domain’s variables. Solving a CSP consists of instantiating each variable with a value in its domain so that the constraints are satisfied.

Our approach to analyzing a MIDI file is divided into the following steps. First, we quantify the MIDI file ~~in order~~ to get rid of slight tempo fluctuations, and we segment it into a series of positions. Then, we define a CSP, whose variables represent the different elements of analysis: [notes (one for each MIDI note-event), chords (at each position), keys (at each position), and melodies], and whose constraints represent the relationships holding between them. The set of constraints depends on the style and the form of the piece. Then we solve the CSP using standard CSP resolution. We use the BackTalk (26) constraint solver to state and solve the problem.

Paleo I/O Formats

Paleo supports compact and efficient data I/O in the form of methods that work with the Squeak Smalltalk **ReferenceStream** framework, a customizable binary object streaming format. The trade-offs in the design of object storage formats are between size, complexity, and flexibility (pick any two). In Paleo, we opted for a system that is compact but also supports the full flexibility of the Smoke music representation, including abstract models for pitch, time, and dynamics, multiple levels of properties and annotation, the attachment of functions of time to events, and hyper-links between events or event lists.

Data files in this format are ~~is~~ on the order of 10–40 times larger than the “corresponding” MIDI files, but because this notation supports the full Smoke annotation, we can store much richer data. Paleo extensions include simple derived properties such as symbolic pitch (with enharmonic disambiguation) and time (with tempo and meter derivation, rest insertion, and

Music and Sound Processing Using Siren

metrical grouping), and higher-level properties such as harmonic analysis, performance expression, and others.

Using Paleo

To set up Paleo, we create a database within a storage directory, then create one or more object sets in it (these correspond to classes or tables), and lastly define indices for the object sets (corresponding to instance variables and accessors). One can then add objects to an object set, or retrieve objects based on queries.

Create a New Database of Scores

The first example establishes a new database and adds an object set to it. The objects we add to this set are assumed to respond to the messages `composer` and `style`. The examples that follow are in Smalltalk; comments are enclosed in double quotes.

```
| dir db |
dir := 'Nomad:Paleo'.           "base directory"
db := SMSDB newOn: dir.        "DB object"
(db addObjectSetNamed: #Scores) "Add an object-set"
  objectsPerFile: 1;
  storesClass: EventList;      "Stores event lists"
                                "Add 2 indices"
  indexOn: #composer domain: String;
  indexOn: #style domain: Symbol.
db save.                       "Save the object set"
                                "Store objects"
db storeAll: (...collection_of_scores...)
```

Make a Simple Query

To make a simple database query, we re-open the database and create a **getOne:** message with one or more **where:** clauses, for example, to get a score by name.

```
| db |
db := MinneStoreDB openOn: 'Nomad:Paleo'.
(db getOne: #Scores)           "Create a query on name"
  where: #name eq: #ScarlattiK004;
  execute                      "Get the first response"
```

Add a New Index to an Existing Database

To add a new index to an existing object set, we use the **indexOn:** message, giving it the name of a “getter” method (i.e., the method that answers the property of the index), or simply a block of Smalltalk code to execute to derive the index value. In the second part of the next example, we create an index of the pitches of the first notes in the score database using a block (the text between the square brackets) that gets the first pitches. This getter block could involve more complex code and/or calls to NOLib functions.

“Add a new index with getter method.”

Music and Sound Processing Using Siren

```
(db objectSetNamed: #Scores)
  indexOn: #name domain: Symbol.

      "Add an index with getter block"
(db objectSetNamed: #Scores)
  indexOn: #firstPitch
  domain: SmallInteger
  getter: [:el | el events first event pitch asMIDI value].
db save.
```

Make a More Sophisticated Query

To retrieve objects from the database, we use **getOne:** or **getAll:** as above and can, for example, ask for a range or the derived first-pitch feature.

```
(db getAll: #Scores)
  where: #firstPitch between: 62 and: 65;
  execute
```

Using Siren for Composition

As stressed at the outset, I develop software tools for use in my own compositions. I documented the use of the earlier MODE system for the piece *Kombination XI* in (8) and (27). My current work in progress is called *Ywe Ye, Yi Jr Di* (abbreviated *YYYJD*) and is based on the text of a Chinese poem of the same name by the great T'ang dynasty poet Du Fu. The sound material for the piece is derived from (a) the voice of a man speaking the text (in a 700-year-old Chinese dialect) and (b) the sound of several small bells.

The piece is written for eight or more channels of surround sound, and the effect should be like that of being inside of a huge collection of different but similar bells that rotate slowly around you and out of which a vocal chant gradually materializes. (The bells are singing to you.)

The bell sounds are processed using a software phase vocoder, a sound analysis/resynthesis package that uses the short-time Fourier transform (STFT) to analyze a sound. One can alter the results of the STFT before resynthesis, allowing, for example, independent control of the pitch and time progress of a sound. For *YYYJD*, I elongate the bell sounds, so that they last several minutes, and transpose their pitches so that I can mix together very dense non-harmonic “chords” based on the bell timbres. Lastly, I apply a slight glissando so that the chords are always decreasing in pitch as they rotate and sing.

For the generation of the bell textures, simple Siren event generators are used to create event lists that are then stored onto text files as programs for the phase vocoder, cmix, and/or SuperCollider software sound synthesis languages (16, 28). I use these external languages for historical reasons; the kind of sound file mixing and layering they do could easily be done in Squeak as well. The main reason for using SuperCollider at present is its ease of programming (its source language and class library are close relatives of Smalltalk), and the fact that it supports the ASIO sound interface, so that one can use eight or more channels of sound output.

Music and Sound Processing Using Siren

Extended EventGenerators and User Interfaces for *YYYJD*

As an example of an extended event generator, I constructed a subclass of **Cluster** called **RotatingBellCluster** for *YYYJD*. This class allows me to easily describe, and then generate, 8-channel textures of inharmonic bell chords where the golden-mean-related “overtones” are placed at separate (related) spatial positions and rotate around the listener at different (slow) rates.

To generate the base sound files (each of which is itself a complex bell timbre), a **RotatingBellCluster** instance can write a command file for the phase vocoder to process a given sound file (a recorded real bell stored as a sound file with 32-bit floating-point samples) making time-stretched transposed copies whose frequencies are related by multiples of the golden mean. This output looks like the following Makefile script for the phase vocoder. The parameters on each line give the time-stretch factors, pitch transposition ratios, and the output file names. This script is run under the shell program on a UNIX compute-server, and often takes many days to process a set of bell samples.

```
# Phase Vocoder script generated by Siren
#   rate fftLen win  dec  int  oscPitchFactor ... inFile  pipe-to      outFile
pv 44100 1024 1024 128 213 0.021286236 0 0 < b.2a.l.float | tosnd -h -f b.2b8.snd
pv 44100 1024 1024 128 213 0.013155617 0 0 < b.2a.l.float | tosnd -h -f b.2b9.snd
pv 44100 1024 1024 128 213 0.008130618 0 0 < b.2a.l.float | tosnd -h -f b.2b0.snd
pv 44100 1024 1024 128 213 1.000000000 0 0 < b.2a.l.float | tosnd -h -f b.2bx.snd
pv 44100 1024 1024 128 213 1.618033988 0 0 < b.2a.l.float | tosnd -h -f b.2ba.snd
pv 44100 1024 1024 128 213 2.618033988 0 0 < b.2a.l.float | tosnd -h -f b.2bb.snd
... many more here...
```

Given a set of sound files, I need a way to precisely describe the way they are mixed and spatialized to generate the desired texture. The basic description of a **RotatingBellCluster** is in terms of two components: (a) the collection of files that ~~comprise~~ make up the texture (each of which is represented by an instance of **BellPartial**), as shown in the first method below, and (b) the three functions that determine the temporal evolution of the bell texture. These functions are shown in the second method below. The basic creation method for **RotatingBellCluster** instances is parameterized with the names of the methods to get these two data sets, so that many different complex instances can be generated and tested.

RotatingBellCluster class methodsFor: ‘B2 Bells’

b2Data

“Answer the data for the b2 series of bell partials.

This array is read and turned into a collection of BellPartial objects.”

“	Name	Freq	Ampl “
^#(‘2ba’	1.6180	15599
	‘2bx’	1.0000	21007
	‘2b1’	0.6180	8560

 Music and Sound Processing Using Siren

```
... other partials included here ...
'2b0' 0.0081 21063 )
```

b2aFunctions

“Answer an array of 3 functions for the bell cluster”

^Array

“Spectral weighting function”

with: (LinearFunction from: #((0 1.5) (0.4 0.8) (1 0.5)))

“Time/density function”

with: (ExponentialFunction from:

#((0 20 -2) (0.3 60 -5) (0.7 50 2) (1 30 0)))

“Mapping of pitch to angular velocity”

with: (ExponentialFunction from: #((0 0 3) (1 1 0)))

The three functions that describe an instance of **RotatingBellCluster** (as given in the method above) are: (a) the relation between the base frequency of a partial and its amplitude (i.e., the virtual spectrum of the summed complex texture); (b) the density of the texture over time (number of partials active at any moment in time); and (c) the relation between a partial’s frequency and its angular (rotational) velocity. The functions defined in the method above are shown in the function editor view in the Figure [below](#).

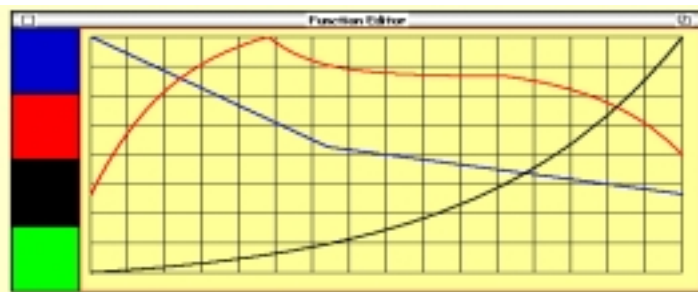


Figure 12: Function Editor on a RotatingBellCluster

To edit **RotatingBellCluster** instances, I use a **TemplateMorph** with a few utility methods to support editing functions in-place and regenerating the event lists on the fly. This editor is shown in [the Figure below](#). The **TemplateMorph** is a simple object editor that lists the “fields” (typically, though not necessarily, the instance variables) on the left and allows one (where meaningful) to edit them in-place on the right.

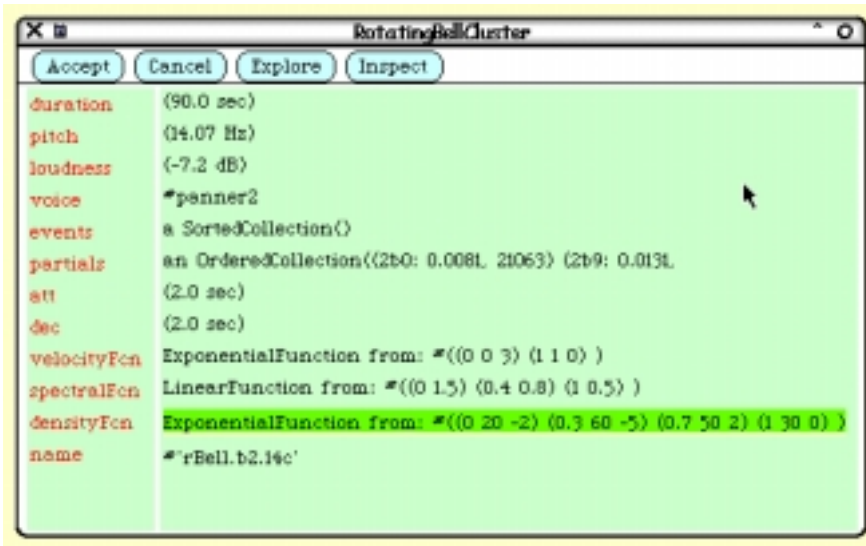


Figure 13: TemplateMorph for Editing RotatingBellClusters

To actually execute this and create a score for doing the sound file mixing, I first wrote a new subclass (which fits easily on one page) of **NoteListVoice** (an **IOVoice**) to support the SuperCollider score file format. This formats the events that are generated by a specific **RotatingBellCluster** and writes out the result as a score file for further processing in SuperCollider. The method that runs this process is shown below, along with an excerpt of the resulting SuperCollider language score file.

```
RotatingBellCluster class methodsFor: 'performance'
run
    "Create, process, and write out a rotating bell cluster from the stored data."
    "RotatingBellCluster run"

    | list scorefile filename |
    list := (RotatingBellCluster
            setup: #b2Data
            functions: #b2aFunctions)
            eventList.
    "Instance creation message"
    "get the event list"

    filename:= 'panner2.sc'.
    scorefile := SuperColliderVoice onFileNamed: filename.

    scorefile play: list; close.

    (FileStream named: filename) edit
```

Music and Sound Processing Using Siren

```
// SuperCollider Score for rotating bell clusters;
//      generated on 28 July 2000 at 5:04:55 am
// NB: #[ is SuperCollider syntax for a literal arra, \x is used for Symbols.
// Instrument command format
// [ delTime , \panner,  \file,   dur,   att, dec,   ampl,  angVel ]
score = #[
  [ 0.5315 , \panner ,  \2bx ,  10.7 ,  2.0 , 2.0 ,  0.8866 , 0.5907 ],
  [ 0.4908 , \panner ,  \2b7 ,  10.7 ,  2.0 , 2.0 ,  1.3346 , 0.0774 ],
  [ 0.4575 , \panner ,  \2b3 ,  10.7 ,  2.0 , 2.0 ,  1.2652 , 0.2520 ],
  ... 414 more lines here ...
];
```

Given these three utility classes (and a couple of methods in **TemplateMorph**), a total of about four pages of code has provided a flexible tool for describing and generating just the kinds of voluminous and complex textures I wanted to serve as the basis of *YYYJD*.

Framework for Linear Prediction

The next phase of production is to take the recorded spoken Chinese voice, turn it into a slow singing voice in a lower pitch range, and then to “cross-synthesize” it with the 8-channel bell textures described above. This involves (many CPU-months of) the phase vocoder mentioned above, and also a linear prediction coding (LPC) vocoder. I use a library for LPC developed for the cmix language (16) by Paul Lansky at Princeton University (parts of which use even older libraries written in FORTRAN!). For this, I linked the library functions I need into a Squeak plug-in that is accessed via a **LinearPredictor** object. An example of the inner-most layer of this **are-is** primitive calls to the plug-in of the following form.

```
LinearPredictor methodsFor: 'analysis'
lpcAnalyze: input out: output rate: rate poles: poles framesize: frsize skip: sk dur: dur
    “Call the CmixPlugin’s LPC analysis primitive.”

    <primitive: 'lpcAnalyzeShort' module:'CmixPlugin'>
    ^self primitiveFailed
```

An example of doing the full LPC analysis is given in the following code excerpt, which reads an input sound file and sets up a linear prediction analyzer to process it. The results are returned in the **LinearPredictor** object, which consists of a series of LPC frame objects.

```
Full analysis example
| sndin data lpc |
sndin := StoredSound fromFile: 'Siren:Kombination2a.snd'.
lpc := LinearPredictor on: sndin.
lpc npoles: 24.
lpc framesize: 256.
lpc analyze.
lpc stabilize.
```

Music and Sound Processing Using Siren

```
lpc packLPCData.
data := lpc pitchTrackLow: 70 high: 500.
lpc packPitchData: data.
lpc explore
```

In the past, I have built fancy GUI tools for manipulating LPC data (27) but now stick to simpler, text-based methods, generally using event generators to generate LPC processing scripts.

Composition Structuring

To manage the larger-scale construction of *YYYJD*, I built a few extensions to existing Siren and Squeak tools. The most useful extension is a refinement to the (excellent) **ObjectExplorer** and **ObjectExplorerWrapper** to support flexible editing of hierarchical event list structures at varying levels of detail. Since Siren events are generic property lists, I need to be able to treat their properties as if they were first-class instance variables and to add new properties and annotations from within the editor. A few pages of extensions to the two classes that make up the explorer made this possible.

The first Figure below-14 shows a simple **SirenScoreExplorer** on the top-level score of *YYYJD*. One can see the hierarchy of the event lists, and the comment of the selected list in the text field below the list. This field has several uses, as a do-it field, an annotation editor, and for adding new properties and relationships.



Figure 14: SirenScoreExplorer on the Structure of YYYJD

This editor can be used to “drill down” to arbitrarily low levels of the structure of a composition, as shown in the following Figure 15, which shows the details of the inner structure of the selected (and expanded) **RotatingBellCluster**.



Figure 15: Details of a RotatingBellCluster

The most powerful extension to the explorer is the ability to select what to show or hide about a given class of objects. The wrappers hold onto lists of fields and support menu selection for varying levels of detail, as illustrated by the two Figure 16s below.

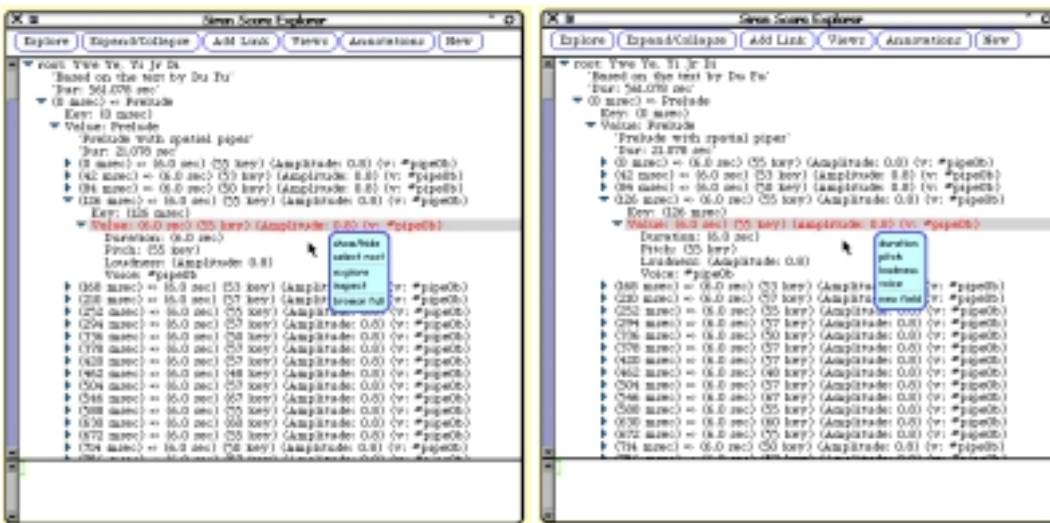


Figure 16: Showing/Hiding Fields in the SirenScoreExplorer

The style of the code extensions and new tools described above are typical of the way I use Siren in a composition. Programming new event generators, new basic music magnitude models, adding script-like classes,

Music and Sound Processing Using Siren

extending the system's graphical tools, and defining new I/O formats are all normal tasks in composition with Siren.

The Future

Status

I have been making my music tools available since 1978. In 1995, I had pretty much given up on using Smalltalk, largely because of the licensing policy of ParcPlace Systems, Inc. Their VisualWorks system was simply too expensive for composers (or even universities) to use, and I was using a bootleg copy myself. (They have since come out with a free-ware version for non-commercial use.) I had invested several months in porting the MODE classes to Java when I heard about Squeak from Dan Ingalls. Because Squeak is open-source, it is a delight that Siren is freely available in compiled (and portable) form. Several components (such as the LPC plug-in and extended MIDI primitives) are, however, platform specific.

At the beginning of the chapter I implied that ~~I~~I am the only Siren customer I listen to; this is not entirely true, but is also not entirely untrue. There have been extended periods where a group of researcher/programmers collaborated on various versions of this software, ranging from the Smoke committee of the years 1990-92 to the more recent Paleo project team at CREATE. I have enjoyed these collaborations immensely, and the results of the team efforts have generally been incorporated into the system.

Current Siren is available from my Web site at CREATE, though the full system only runs well on Apple Macintosh computers (see below). All of the base Siren system, as well as the SMS database framework and most of the Paleo code, is included.

Successes

It has been a thrill to develop tools for my own use in Smalltalk and to find that they are sometimes of use to other composers. Since what I do is already a pretty elitist effort (contemporary serious music), ~~I~~I am always ~~thrilled~~ pleased when someone actually uses Siren and sends me comments (or a bug report). It has been interesting to see what features external users add to the system, and a number of these have been incorporated recently (e.g., Alberto De Campo's microtonal MIDI representation).

Failures

Most of the disappointments for Siren's users (and me) are based on the basic profile of what Siren is intended to do in the first place. I have only used MIDI for one experimental composition (that is now played from tape in concerts), and I am not a big fan of graphical tools, so many users are disappointed that Siren is not a free MIDI sequencer, a graphical programming language for music, or an object-oriented sound synthesis system. (All of these are available elsewhere.)

While it has been very fulfilling to have active collaborators, there have also been frustrations, such as the difficulty of achieving multi-platform support for a standard set of high-level MIDI I/O functions. The base set of

Music and Sound Processing Using Siren

primitives included in Squeak is inappropriate for intensive MIDI-oriented applications, and there appear to be no volunteers to maintain a set of more powerful primitives on multiple platforms.

Siren 2002

When I look ahead to the future, I hope that in two years I have: more of the same only faster and more flexible! My main wishes are related to the hardware I use, rather than weaknesses in Squeak or Siren. (I've given up on ever getting the 32-hour day.) That being said, the main features that I miss in Squeak can be grouped as follows:

Language issues: multiple value returns and multiple value assignment, true multiple inheritance (to be used rarely, of course), assertions, multiple kinds of comment characters;

Library issues: CORBA IDL and ORB integration, probability distributions (a la ST80), very high-precision floating-point numbers, integrated namespaces and packages;

Tool issues: integrated source code control system and configuration management tools, tools for CORBA distributed applications; and

GUI issues: native look and feel, themes.

My primary fear about Squeak is related to "image bloat." The system is growing fast, and we have still not integrated namespaces and packages to the point where image management is easy.

My plans for the next few months (as I work on *YYYJD*), are to integrate the Steinberg ASIO sound libraries (as soon as it's clear what will be supported on the next generation of Macintosh operating systems), and to incorporate more of the LPC vocoder functions into Siren. Medium-term, I intend to work on a more powerful synthesis engine (stealing ideas from SuperCollider) and to integrate Siren and SuperCollider as plug-ins to each other. Lastly, I have already composed the follow-on piece to *YYYJD* (the acronym is *TMYB*) and hope to start working on it in early 2001.

Acknowledgments

It would be extremely unfair to present this work as mine alone. Siren incorporates the work of many people who have contributed ideas and/or code; they include: Paul Alderman, Alberto De Campo, Roger Dannenberg, Lounette Dyer, Adrian Freed, Guy Garnett, Kurt Hebel, Craig Latta, David Leibs, Mark Lentczner, Hitoshi Katta, Alex Kouznetsov, James McCartney, Hans-Martin Mosner, Danny Oppenheim, Nicola Orio, Francois Pachet, Pierre Roy, Carla Scaletti, Bill Schottstaedt, John Tangney, and Bill Walker. Pierre Roy and Nicola Orio also contributed texts to this chapter.

I must also here acknowledge the generous support of my employers and the academic institutions where this software was developed, including PCS/Cadmus GmbH in Munich, Xerox PARC, ParcPlace Systems, Inc., CCRMA/Stanford, the STEIM Foundation in Amsterdam, The Swedish Institute for Computer Science, CMNAT/Berkeley, CREATE/UC Santa Barbara, and the electronic music studio of the Technical University of Berlin. Special thanks for moral and financial support during the writing of this chapter go to CREATE's director JoAnn Kuchera-Morin and the manager of

Music and Sound Processing Using Siren

the TU Berlin studio, Folkmar Hein, as well as to the Deutscher Akademischer Austauschdienst (DAAD).

Lastly, Mark Guzdial and Juan Manuel Vuletich read and provided very useful comments on an earlier draft of this text.

Conclusions

Siren is a framework in Squeak Smalltalk for music and sound processing. The main focus is towards the representation of structured music and musical composition methods. This chapter described the Smoke representation language that serves as the kernel of Siren, introduced its I/O facilities, and presented several extended examples of Siren in action.

References

1. Ingalls, D., T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself." *Proc. ACM OOPSLA 1997*.
2. Pope, S. T. "Bat Out Of Hell" (musical composition) in *Perspectives of New Music* V. 24, 1986 (cassette). Also in *Computer Music Journal Sound Anthology*, V. 21, 1997 (CD).
3. Pope, S. T. "Music Notation and the Representation of Musical Structure and Knowledge." in *Perspectives of New Music* 24(2), Winter, 1986.
4. Pope, S. T. "The Development of an Intelligent Composer's Assistant: Interactive Graphics Tools and Knowledge Representation for Composers." in *Proc. 1986 International Computer Music Conference*. San Francisco: International Computer Music Association.
5. Pope, S. T. "A Smalltalk-80-based Music Toolkit." in *Proc. 1987 International Computer Music Conference*. San Francisco: International Computer Music Association.
6. Pope, S. T. "The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers." *Computer Music Journal* 16(3) (Fall, 1992).
7. Pope, S. T. "An Introduction to the MODE." in (11).
8. Pope, S. T. "Kombination XI" (musical composition) in *Or Some Computer Music*. Touch/OR Records, 1999. Also in *The Virtuoso in the Computer Age* (CDCM V. 13). Centaur Records, 1993.
9. Pope, S. T. "Siren: Software for Music Composition and Performance in Squeak." *Proc. 1997 International Computer Music Conference*. San Francisco: International Computer Music Association.
10. Pope, S. T. "The Siren Music/Sound Package for Squeak Smalltalk." *Proc. ACM OOPSLA 1998*.
11. Pope, S. T., ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. MIT Press, 1991.
12. Roads, C., S. T. Pope, G. DePoli, and A. Piccialli, eds. *Musical Signal Processing*. Swets & Zeitlinger, 1997.

 Music and Sound Processing Using Siren

13. Dannenberg, R. "Music Representation Issues, Techniques, and Systems" *Computer Music Journal* 17(3), Fall, 1993.
14. Wiggins, G. et al. 1993. "A Framework for the Evaluation of Music Representation Systems" *Computer Music Journal* 17(3), Fall, 1993.
15. Pope, S. T. "Musical Object Representation." In (12).
16. Pope, S. T. "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal* 17(2), Summer, 1993.
17. Scaletti, C. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2), Summer, 1989. reprinted in (11).
18. Pope, S. T. "Modeling Musical Structures as EventGenerators." In *Proc. 1989 International Computer Music Conference*. San Francisco: International Computer Music Association.
19. Dannenberg, R., P. Desain, and H-J. Honing. "Programming Language Design for Musical Signal Processing" in (12).
20. Krasner, G., and S. T. Pope. "A Cookbook for the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1(3), 1988.
21. Pope, S. T., N. Harter, and K. Pier. "A Navigator for UNIX." *1989 ACM SIGCHI video collection*.
22. Carlson, J. 1998. MinneStore OO Database Documentation. See <http://www.objectcomposition.com>.
23. Huron, D. ~~1994~~. "The Humdrum Toolkit Reference Manual,". Center for Computer Assisted Research in the Humanities, Menlo Park, California, 1994.
24. Mouton, R. ~~1994~~. "Outils intelligents pour les musicologues," Ph.D. Thesis, Université du Maine, Le Mans, France, 1994.
25. Mackworth, A. ~~1977~~. "Consistency in Networks of Relations,". *Artificial intelligence*, 8-(1), pp. ~~99-118~~1977.
26. Roy, P., A. Liret, and F. Pachet. "The Framework Approach for Constraint Satisfaction," *ACM Computing Survey Symposium*, 1998.
27. Pope, S. T. "Producing 'Kombination XI:' Using Modern Hardware and Software Systems for Composition." *Leonardo Music Journal*, 2(1~~-~~), 1992.
28. McCartney, J. SuperCollider Software. See www.audiosynth.com.