

Part 1. Introduction

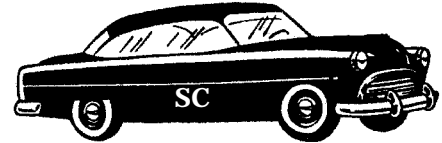
The SuperCollider Language and System

SuperCollider is a powerful and flexible programming language for sound and image synthesis and processing. It was developed by James McCartney of Austin, Texas, and is the result of more than five years of development, including the Pyrite and Synthomatic systems from which SuperCollider is derived. The somewhat odd name of the language is derived from its creator's obsession with the superconducting supercollider project that was planned to be undertaken in his home state of Texas, but never funded.

The SuperCollider compiler and run-time system has been implemented on Apple Macintosh and Be computers (more ports are projected), and can execute quite complicated instruments in real time on "middle-class" Macintoshes (see the notes below on its performance). This book is a step-by-step tutorial on SuperCollider programming; it is aimed at musicians who want to use it for musical sound synthesis and processing.

SuperCollider and Music-V-style Languages

SuperCollider (hereafter SC) is a sophisticated high-level programming language; its syntax and library functions are derived from the C++ and Smalltalk languages. Its development environment includes a program text editor, rapid turn-around compiler, run-time system, and graphical user interface (GUI) builder. SC instruments can also take their parameter inputs from real-time MIDI commands and controllers, and can process sound files and live sound input.



The main differences between SC and "traditional" Music-V-style software sound synthesis languages such as cmusic or Csound are (1) most SC programs run in real time and can process live sound and/or MIDI inputs; (2) SC is a comprehensive general-purpose programming language with facilities for file input/output (I/O), list processing, and object-oriented programming; and (3) SC is an integrated programming environment including a text editor and GUI builder that allows you to build interactive interfaces for your instruments.

Reading This Book

This book is an introduction to the SC language aimed at readers who have some programming background (such as knowing another sound synthesis language or a general-purpose language like C or Smalltalk). It is not meant to substitute for the SC manual, to which I indeed refer the reader in numerous places.

For a tutorial on computer music, readers should consult a general text book such as Curtis Roads's comprehensive book *The Computer Music Tutorial* (Roads 1996) (highly recommended). For more in-depth programming techniques (in C) and engineering details, I suggest *Elements of Computer Music* by F. R. Moore (Moore 1991). If you have no programming background at all, you should take some time to learn the basics of the C, Pascal, LISP, or Smalltalk programming languages before proceeding (e.g., Deitel and Deitel 1992).

The recommended way to use this book is to (1) get access to a computer capable of running SC (an Apple PowerPC-based Macintosh, third-party Macintosh-compatible clone, or BeBox), (2) install SC on it (either the free demo version [see Part 8] or the full [paid-for] installation will work), and (3) read through the book consulting the reference manual and *Computer Music Tutorial* where appropriate, and using the on-line example code.

The documented source code for all examples presented in this book is available by Internet FTP or WWW access from the CREATE Web site at UCSB; look at the Web page with the URL <http://www.create.ucsb.edu/SC/> or the FTP directory [ftp.create.ucsb.edu/pub/SC](ftp://ftp.create.ucsb.edu/pub/SC). The effectiveness of this book is greatly enhanced by on-line experimentation with the progressive examples it presents.

This Part introduces the topic and presents a very simple example to give readers the taste of SC. If you have no background in software sound synthesis, I suggest that you skip to Part 2. It presents the background of sound synthesis programming languages, especially the Music V family. Part 3 introduces the syntax of the SC programming language and its rich collection of built-in functions. In Part 4, the SC user interface components are outlined. I present the basic programming techniques of SC in Part 5. Following that, a collection of advanced techniques available to SC programmers are demonstrated in Part 6. The last major topic is algorithmic composition, which is discussed in Part 7. Part 8 presents some final comments and information on where to get SC itself and more information about it.

Acknowledgments

My first acknowledgment must go to James McCartney for creating SC in the first place. It is only because I like using it so much—and also believe that it is an excellent production, performance, and pedagogical tool—that this book ever came to exist. A number of people at CREATE have contributed to this effort, but I would especially like to thank Curtis Roads for his support and input, and the students of Music 209L at UCSB for their questions. CREATE's director, JoAnn Kuchera-Morin was also central to

the process because she worked for 12 years to establish an environment that fosters artistic experimentation and production without compromises, and attracted the researchers and students with whom I feel lucky to work.

Typographical Conventions

In this book, all program code is written in the Helvetica typeface. Jargon terms are written in *italic style*. Menu items and keyboard commands are written in **bold-face**.

A Simple Example

As a quick taste for the impatient, the example below is an annotated SC instrument that plays a continuous sine wave at 220 Hz. Read the comments in the right column first (“--” starts a comment that continues up to the end of the line). Because this is a real-time system, we do not need an envelope or even start/stop times. The instrument plays as soon as it is compiled and executed within the SC environment.

To run this example, start up SC on your computer and find the example code that accompanies this book. Open the file “01. Steady Oscillator” in the “Code” folder; you’ll notice that the text is displayed in SC’s text editor view. To play the instrument, press the **Command-/** key combination); the sound will continue until the program is interrupted (by pressing the **Command-.** “interrupt” keys). To make it softer, use the **Command-]** keys; **Command-[** makes it louder.

```
-- A Simple SuperCollider Oscillator "Instrument" Program (This is a 1-line comment.)
--
defaudioout L, R;                                -- Define audio outputs named L and R.

start {                                           -- The "start" function is called automatically (like
-- "main" in C programs); it usually runs the
-- instrument. In this case, it is the instrument.
    var osc, outval;                             -- Declare variable names (with no type information)
-- for the oscillator object and output sample buffer.
    osc = Asinosc(220, 0);                       -- Create a 220 Hz sine oscillator object.
-- Asinosc()'s arguments are (frequency, phase).
-- The oscillator object is held in the variable "osc."

    dspAdd({                                     -- Now create a continuous loop to play the oscillator.
-- Start the signal processing loop.
        outval = value(osc);                     -- Get osc's value (a buffer full of samples).
        out(outval, L);                          -- Send it to the left output.
        out(outval, R);                          -- Send it to the right output.
    });                                           -- End the DSP loop.
}                                                 -- That's it! (End the start function, and the program.)
```

Code Example 8. An Introductory Example

This simple example already has all the standard elements of SC programs: a header comment, sound output declarations, and a start function with a unit generator construction message and a DSP loop that evaluates the unit generator.

The details of the SC language syntax and its library functions will be presented in Part 3, and Part 5 includes a comprehensive set of progressive examples.

SuperCollider Summary

As in most languages of the Music V family, SC has built-in functions for various kinds of sound sources (oscillators, noise generators, sounds file readers, live inputs, etc.), time functions (envelopes, controls), sound processors (filters, delay lines, etc.), and outputs (to write sound samples to the digital-to-analog convertor or to a file).

The two most important differences between SC and the Music V family are that SC always tries to run in real time, and that it can read live audio input and MIDI data. SC is also a “real” programming language in that it has more comprehensive data types (e.g., lists) and built-in functions (e.g., text file-I/O and graphics) than other “sound compilers.”

The rest of this book is about SC, but first, we will introduce the basic technology of its predecessors—the Music V family of software sound synthesis (SWSS) programming languages.

SuperCollider Performance

While meaningful benchmarks of software sound synthesis systems are hard to design (Pope 1993), SC is unique in being strongly oriented toward real-time performance, and using quite sophisticated compiler and run-time execution technology. The user interface gives you constant feedback during performance about how much of the CPU is being used, so you can always tell if you are close to being “in trouble” in terms of real-time performance. SC also allows you to compute samples out of real time and write them to disk for later performance. (It saves the sound files in SoundDesigner II format.) Thus—as with traditional software sound synthesis languages—there is no real limit on the complexity of instruments you can write in SC (if you’re patient enough).



There are several variables that effect run-time performance. The most obvious of these is the *output sample rate* you use. The “standard” rate is 44100 Hz (the CD sampling rate), but values between 11025 and 48000 Hz are meaningful. A lower rate means more computation can be done per sample, but also limits the frequency response of the output. Like several other sound synthesis lan-

guages, SC computes envelopes and other “control” functions at a lower rate than the sample rate. The “sub-frame” size is 64 by default, meaning that control functions are updated every 64 samples. This can be set to any size between 4 and 256 (it’s usually a power of 2); smaller sub-frames cause more computation (and better sound). The third variable that effects computation speed is the *output buffer size* (or “frame size”, which determines how often sample buffers are sent to the Macintosh sound manager. The default is 4096 samples; large output buffers improve run-time performance, but impede interactivity in instruments with run-time control inputs.

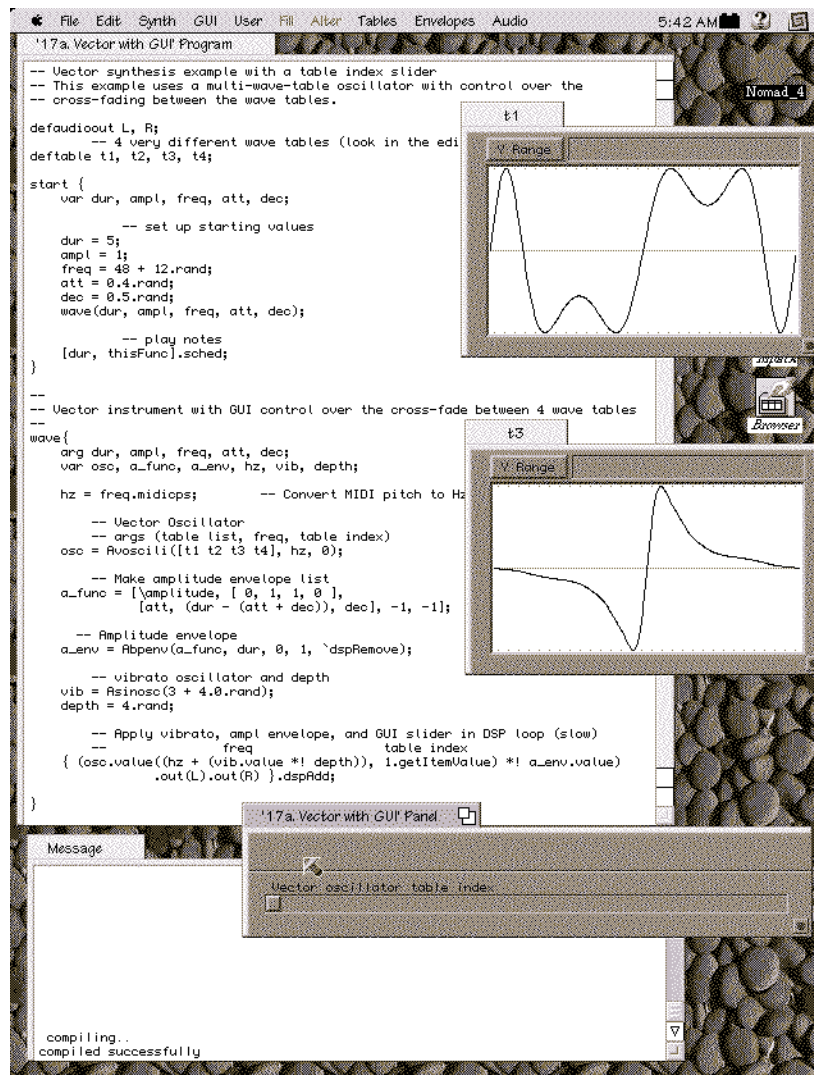
The machine on which I’ve used SC most heavily is an Apple Macintosh PowerBook 1400cs (with a 183 MHz PPC 603e CPU, 128 kB cache, and 28 MB RAM). On this machine, the sine instrument given above takes approximately 9% of the CPU resources, while the FM instrument introduced below (with parameterized envelopes for the amplitude and modulation index, and a repeating note pattern) takes about 22%.

These performance values are measured with the default settings of 44100 Hz sample rate, a control buffer (sub-frame) size of 64 samples, and an output buffer (frame) of 4096 samples; these three variables can all be tuned to improve performance. If we decrease the sample rate to 22050 Hz and increase the sample buffer size to 256 samples, for example, the FM program uses less than 13% of the CPU on the PowerBook (and obviously sounds different).

A score that uses four simultaneous voices of the FM instrument takes 95% of the CPU using the default settings; increasing the output frame size to 16384 samples brings this down to 89% (because of the decreased sound manager overhead). Changing the sample rate to 22050 Hz means that it takes 51%, but it goes back up to 89% if you decrease the sub-frame size (increase the control rate) to 16 samples at the same sample rate. The table below illustrates the CPU usage for various settings of the three relevant variables and gives some hints about how to “tune” instruments for best performance.

Sample Rate	Sub-Frame Size	Frame Size	% CPU Usage
44100	64	4096	95
44100	64	16384	89
44100	256	4096	70
22050	64	4096	51
22050	16	4096	89
22050	512	8192	35

Table 1: Performance of a four-voice FM instrument



Part 2. Software Sound Synthesis

2.1. Historical Introduction to Software Sound Synthesis¹

Computer music as a field has been likened to a building with a sign on it saying “Best Eats in Town.” Many people go into this building expecting to find an elegant restaurant with a parchment menu, formidable wine list, and pleasant, efficient, even charming service. What they find instead, to their surprise, is a shiny, enormous, extremely modern kitchen, with abundant supplies of every kind of foodstuff in voluminous, refrigerated storage. Indeed, the “Best Eats in Town” are available here, but only to those willing to learn to cook! (Moore 1983)

There have historically been four distinct classes of electroacoustic music instruments or systems: the early electronic instruments of the pre-WW II era, the tape-based *musique concrète* studios; modular analog synthesizers of (e.g.) Moog, Buchla, ARP, and EMS; and software-based sound synthesis (SWSS) systems as described in the landmark book *The Technology of Computer Music* (Mathews 1969), which described his Music V sound synthesis program developed at Bell Telephone Laboratories. The technology of Music V-style languages (often referred to as *MusicN* languages) will be described below. A major factor in the wide use of Music V during the 1970s was the fact that it is written almost entirely in FORTRAN (a programming language that ran on many kinds of computers); its predecessors were generally written in assembly language, and were thus not portable among machine architectures.

This technology was widely used during the 1970s in the form of several Music V descendents that ran on the mainframe computers of the day. The 1980s saw a further steady increase in the availability of SWSS systems in the form of systems based on DEC PDP-11 computers using Barry Vercoe’s Music-11, and later DEC VAX-11 machines running the CARL/cmusc system developed by F. Richard Moore and D. Gareth Loy at the University of California, San Diego.

More recently, we have seen the rise of the “computer music workstation,” including diverse configurations based on personal computers and digital signal processing (DSP) subsystems, and of course, the plethora of less flexible but real-time-capable MIDI-based computer music systems. Figure 1 below shows a partial genealogy of the

1. This chapter is a revised excerpt from the author’s article “Machine Tongues XV: Three Packages for Software Sound Synthesis” that appeared in *Computer Music Journal* 17(2): 23-54.

MusicN languages and related systems—defined as those based on a software implementation of the unit generator instrument model and the function/note list score model. The *Computer Music Journal* article (Pope 1992) presents an in-depth discussion of the engineering aspects of computer music workstations using modern technology, and introduces several of the systems listed in the graph below.

The Early SWSS Literature

The SWSS literature is generally said to have started with the abstract and publication of the “Acoustical Compiler” article by Max Mathews (1960, 1961) in the *Journal of the Acoustical Society of America* and the *Bell Laboratories Technical Journal*. The field gained much more visibility with the more widely circulated articles (Mathews 1963)—a popular and introductory essay—and (Tenney 1963)—a comment on the genesis of the field from a composer’s point of view. The state of the art at the time of the development of Music V, in addition to the early experiments undertaken using it, are described in (Pierce, Mathews, and Risset 1965).

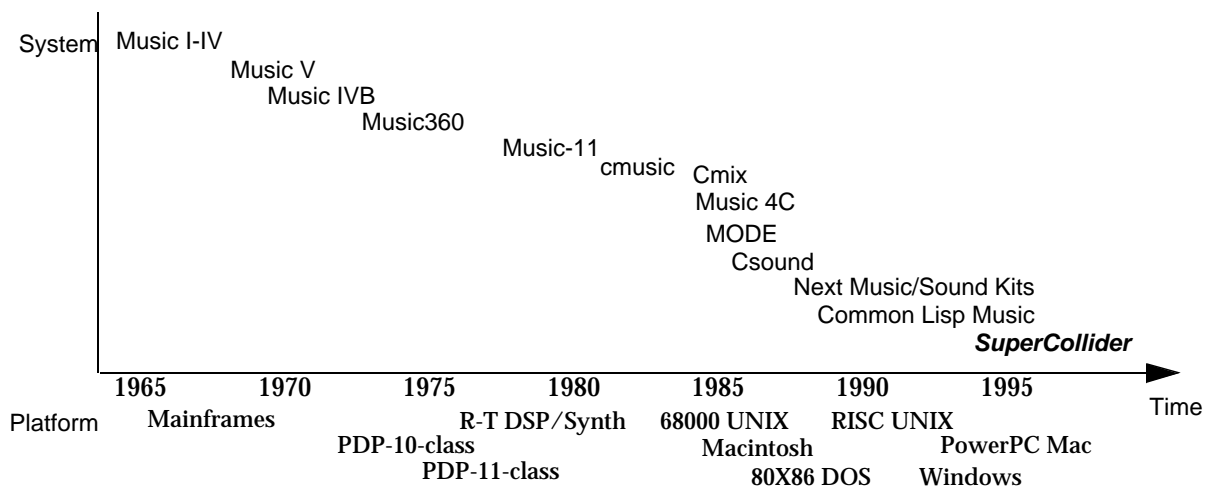


Figure 1. Timeline of MusicN SWSS Systems

Recent SWSS Packages

There are a number of other MusicN SWSS languages in common use today; the cmusic (Moore 1991), cmix (Lansky 1994), and Music 4C (Beauchamp 1989) languages are all based on the C language; Barry Vercoe’s Csound (Vercoe 1996) takes assembly language as its model. Bill Schottstaedt’s Common Lisp Music, or clm (Schottstaedt 1992), and this author’s MODE Musical Object Development Environment (Pope 1992) are

based on (and written in) LISP and Smalltalk, respectively. All of these provide a high-level score input language, some graphical tools, and easy-to-use languages for building extended synthesis functions.

2.2. The Technology of Computer Music

With reference to the design of Music V, Max Mathews wrote:

The two fundamental problems in sound synthesis are (1) the vast amount of data need to specify a [sound] pressure function—hence the necessity of a very fast and effective computer program—and (2) the need for a simple, powerful language in which to describe a complex sequence of sounds. Our solution to these problems involved three principles: (1) stored functions to speed computation, (2) unit generator building blocks for sound-synthesizing instruments to provide great flexibility, and (3) the note concept for describing sound sequences. [...] [The composer] would like to have a very powerful and flexible language in which he/she can specify any sequence of sounds. At the same time, he/she would like a very simple language in which much can be said in a few words, that is, one in which much sound can be described with little work. [...] In a given instrument, the composer can connect as many or as few unit generators together as he/she desires. (Mathews 1969 p. 34-5)

A sound structure is programmed in two parts in MusicN languages—the *instrument definition* describes the connections of signal generators and modifiers for the timbres that are to be used, and the *note list* is the score, described in terms of parameters sent to the instruments. The synthesis model is similar to that of an traditional analog synthesizer. One makes a “patch” among modules such as oscillators, amplifiers, mixers, and control function generators (the so-called *unit generators* or UGs of SWSS programs), and then sends trigger and control data to the patch to make sounds.

The parallels between the models of unit generators and the modular analog synthesizers developed in the same years have obvious advantages. They allow many composers to map their prior experience into the new realm (assuming that most composers coming to computer music have used analog synthesizers, which used to be true), and to make structured, scalable instrument descriptions. The disadvantage is that many types of sounds (e.g., those based on complex time-varying filters), are not easily modeled in terms of simple patches of the standard unit generators.

Early SWSS systems were “monolithic” in that they were not accompanied by DSP programs that could be used independently of the sound compiler. It was assumed that a composer would express an entire piece as a note list, and that all synthesis and pro-

cessing would be done within the instruments. Because of limited disk storage, and the inflexibility of magnetic tape storage, it was often not possible to use recorded sound or to process synthetic sounds in multiple stages.

Since the advent of the CARL software system in the early 1980s, though, most SWSS systems include a MusicN compiler program as well as a suite of “stand-alone” DSP tools that can read and write recorded or synthetic sound files. This allows the composer to create his/her basic sounds with a SWSS system (such as SC), and then to use other tools for mixing, reverberation, or post-processing. One can also use several passes of a sound compiler such as SC for creating, mixing, and processing sounds, with the intermediate sounds stored on disk. The Macintosh is an excellent platform for this kind of multi-tool production because of the wide variety of software for sound synthesis and processing. At CREATE, our users integrate SC together with the Deck mixing program, SoundHack for many kinds of DSP, HyperPrism for interactive control of processing, and several other tools.

Instrument Definitions

There are strong parallels between a traditional SWSS *orchestra* file and a typical program’s source code; it includes some header information—such as the sampling rate and output file format— (like the program’s header and declarations), which is followed by one or more instrument definitions (like subroutines in a program). The *score* file or note list initializes certain shared data, e.g., function wave tables, and then contains a list of note commands that activate the orchestra’s instruments at stated times with given parameters (like a batch data processing input file). The result of executing a SWSS program—running the sound compiler with the source and data files—is a (possibly huge) output file of digital sampled sound, which can be listened to using a *play* program to send it to the output digital-to-analog convertors (DACs) of the system in real time. In SC, this data is played while it is being generated, but you can store it to a disk file if you wish. An instrument definition is structured like a subroutine, macro, or procedure definition in any standard programming language (e.g., PASCAL, or C); there are variable declarations, set-up expressions, and a repeated loop of data manipulation statements that write into one or more output buffers. Examples of this structure will be presented below.

In the process of realizing compositions, SWSS users sometimes develop very many instruments. This orchestra may consist of variations of several common models (e.g., frequency modulation [FM] or sound-file processing), and instruments may range from the very general—having many parameters and a wide range of musically-useful appli-

cations—to the very specific—having few parameters and a more concrete (less customizable and possibly more dynamic) musical gesture. Instruments may generate output based solely on their input parameters (as in traditional oscillator-based instruments), or they may read real-time control data (e.g., via MIDI), or process pre-existing sound files (as in filtering or mixing instruments).

A number of graphical representations and visualization tools for instrument definitions have been used for MusicN languages. The most common ones use a flow-chart or data-flow style diagram to show the signal flow among unit generators where instruments generally “flow down” from input parameters through control signals to audio signals to the output. Data-flow block diagrams with (generally multi-input single-output) graphical icons representing instrument unit generator modules and connecting lines or arcs representing parameters or control or signal buffers will be used throughout the discussion of SWSS instruments below. A single statement (line) in the instrument definition program will often translate into a single block icon whereby the arguments of the statement determine the connections between the icon’s control and signal I/O ports.

Oscillator Unit Generators

The most basic unit generators in SWSS systems are stored-function oscillators—sub-routines that read data values out of a table stored in memory (the envelope function or wave table), at a rate that is computed using a formula relating the size of the table, the sampling rate, and the frequency of the desired sound signal or duration of the control envelope (see [Roads 1996] for details). An oscillator statement generally includes the command name (OSC, osc, oscil, etc.), the amplitude value (constant or function), the frequency value (constant, envelope or audio-rate signal), the wave table name or number, and the output buffer name. The order and format of the parameters differs among SWSS systems, but the four basic parameters—output, amplitude, frequency, and wave table—remain the same.

Function Generators

To create function tables for use as envelopes or wave-forms, generator commands fill data tables with values that depend on their parameters and which routine they use. There are usually several ways—GEN routines—to describe such vector data in SWSS languages, such as by interpolation between break-point values (as in sound envelopes), by the summation of related sinusoidal components (as in wave table generation for additive synthesis), or by reading in external data files with or without some analysis and feature extraction.

Envelope Unit Generators

SWSS systems have several ways of providing the functionality of *envelope generators*, unit generators that produce functions of time that step through a table once per note. This can be achieved by setting an oscillator's sample increment to depend on the inverse of the length of the desired note, rather than the output frequency (i.e., read through the function table once per note duration). In general, one can define line-segment, or exponential-segment functions, or use a stored envelope function, and read through them with control over the speed of the sections, usually used to control the attack and decay times of envelope functions.

Other Unit Generators

Modern SWSS systems provide many low- and high-level control and audio signal generator and modifier unit generators. These may include (e.g.,) variable wave form oscillators, noise and pulse generators, sound file input unit generators, multi-segment envelope generators, digital filters, digital delay lines, or other musical, or DSP functions. Some manner of output unit generator is also required; this will read one or several instrument buffers and write to formatted sound files, or unformatted sample streams. Some systems add room simulation unit generators allowing the user to declare a spatial configuration for a room and position sound sources in it.

Score Note Lists

The statements that describe which notes the instruments are to play, how, and when, are the *note list* part of the SWSS music description. This score file includes the set-up of the function tables using GEN statements, the declaration of the input and output sound files and their formats, and the time-stamped note event data—a list of expressions that activate the instruments one-by-one at specified times with parameters supplied in the statement. The note statement used for this consists of its keyword (NOTE, not, instr, etc.), the start time and duration of the event, the instrument number (or name), and the parameters of the instrument (e.g., amplitude, frequency, location, timbral properties).

There are generally facilities to use abstract time notations in SWSS scores, with the score's tempo defined as a beat-to-second map, and some way of setting and changing it. Most score languages also allow longer scores to be broken up into sections, each of which can have a separate clock and tempo. The purpose of sections is that they are sorted separately, and each start at relative time 0. The sections are computed in sequence by the program's scheduler. SWSS languages often also provide powerful facilities for generating and structuring note list files, and various kinds of short-hand

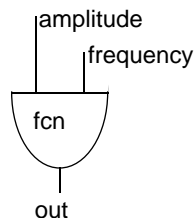
for making the input and management of (possible very complex) instrument parameter field data for larger musical forms less laborious.

2.3. An Example of Software Sound Generation

This section presents examples of Music V using a simple instrument. We will discuss the instrument definition format, the score list syntax, and the process of executing the “compiler” package for such a system.

A simple unit generator and an instrument definition patch are shown in Fig. 2; Fig. 2(a) shows the graphical symbol for an oscillator unit generator with its four relevant features: amplitude, frequency (or sample increment), wave form (timbre), and output. Figure 2(b) illustrates the use of this unit generator in an instrument; two oscillators are connected such that the inputs of the first one (the “envelope generator”) control its amplitude and its sample increment based on the note’s duration; its output is connected to the amplitude input of a wave table oscillator that has its sample increment derived from the note’s frequency (the oscillator’s repetition rate). The first oscillator thus functions as an envelope generator that generates a time-varying control function for the amplitude envelope of the second oscillator’s notes. The parameters of this instrument are the pitch, amplitude, and wave form of the output signal oscillator, the duration of a note, and the amplitude envelope function.

(a) Osc unit generator



(b) Oscillator with amplitude control

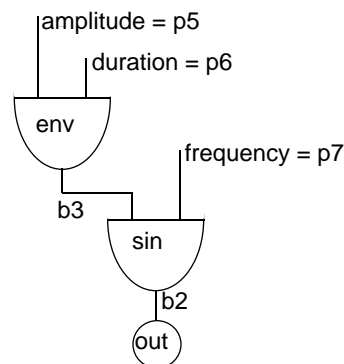


Figure 2. The Oscillator Unit Generator (a) and a Simple Instrument Patch (b)

In order to use this instrument, one must write a program that defines three components: the instrument, the function tables for the two lookup oscillators, and the desired

note parameters. There will be two function tables: one holding values for the amplitude envelope (e.g., Fig. 3(a)) that will be read through once per note, and the other describing the output oscillator's wave form function (e.g., Fig. 3(b)) to be read through at a rate that depends on the frequency of the note. The commands that declare and define function tables may be seen as being part of the instrument or the score, depending on the nature of the compile-run cycle (see below).

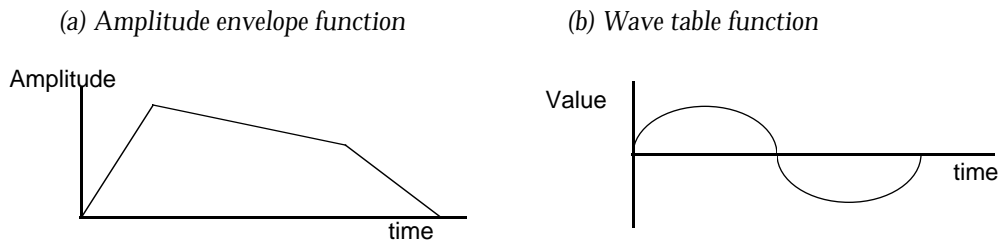


Figure 3. Function Table Values for an Envelope and a Wave Form Signal

The Music V instrument definition for this example would read as shown in the first group of statements in Code Example 2. Comments start with the COM statement and go until the end of the line in Music V. The instrument definition uses the OSC unit generator for both the envelope function and the wave form oscillator by having them read different function tables. Note the use of two different buffer numbers for the signal buffers here; in some SWSS systems, one can reuse the same buffer number in several places in an instrument, as is possible here because no unit generator depends on *both* b1 and b2 for its input. The Music V OUT unit generator writes samples from its input to its standard output buffer B1, which is written to the output disk.

The specification of unit generator parameters and the translation of note command parameters varies greatly among SWSS systems. In Music V, for example, one typically passes amplitude and frequency parameters directly from the score to the unit generators, and is forced to translate from pitch values to oscillator sample increments and from loudness values to integer amplitudes in the score.

```
COM    Music V Instrument
COM    Note Parameters are:
COM        p2=start, p3=instr_num,
COM        p4=duration, p5=ampl,
COM        p6=dur_incr, p7=freq_incr
```

```

COM   Definition for instr. 1
      INS 0 1;

COM   The first oscillator is the amplitude envelope; the second is the audio signal.
COM       AMP  FRQ  OUT  FCN  TMP
      OSC P5   P6   B3   F1   P20;
      OSC B3   P7   B2   F2   P21;

COM   Send buffer 2 to the output.
      OUT B2 B1;
      END;

COM   Generate Function Tables

COM   Generate function 1 as an envelope with GEN 1
COM   Routine 1 takes x/y breakpoints
      GEN 0 1 1  0 0  0.99 50  0.8 480 0 511;

COM   Generate function 2 as a sine using GEN 2
COM   Routine 3 takes partial num. and ampl.
      GEN 0 2 2  1 1;

COM   Play Two Notes

COM p1 p2 p3 p4 p5   p6   p7
      NOT 0 1  2  30000 0.0128 6.70;
      NOT 2 1  4  8000  0.0064 8.20;

COM   Terminate the score at time 6
      TER 6;

```

Code Example 9. Music V Instrument Definition and Note List

A Music V score file for this example instrument would first define the two function tables and then play notes on the instrument by providing values for its parameters, as in the second group of statements in Code Example 2. One declares function tables 1 and 2 using the GEN command and routines 1 (linear interpolation between breakpoints), and 2 (summation of sines). The parameter data in the note command p-fields signifies the notes' parameters—start time, instrument number, duration, amplitude, envelope duration increment, and oscillator frequency increment. The amplitude is given in absolute numbers (assumed in the range 0-32767 for this example, implying 16-bit linear samples). Note the increment parameters; p6 is the envelope increment—related to the table length, the inverse of the note duration, and the sample rate—p7 is for the frequency increment—related to the frequency, the table length, and the sample rate.

There may be one, a few, or many notes in the subsequent note list, depending upon whether the user is testing the instrument's parameters, developing small musical textures or gestures, or performing an entire section or composition in the current "pass." Notes can overlap, and several notes may be active in the same instrument at the same time; the system's scheduler handles multiple instrument activations and output summation.

To execute a Music V program, the instrument definition (possibly defining many instruments), is read, and possibly compiled into a compact and efficient internal format; then the note list is expanded and sorted, possibly applying score language pre-processors. The actual sample computation task consists of a "scheduler" reading through the score data in time order, activating instruments as appropriate, and summing their respective outputs into the output sound file(s). One "plays" the sounds using a program that sends the sample data (stored on disk or tape) to a DAC in real time. (In the early days, this was often a different machine from the one that computed the samples.) The steps of the process are illustrated in Fig. 4.

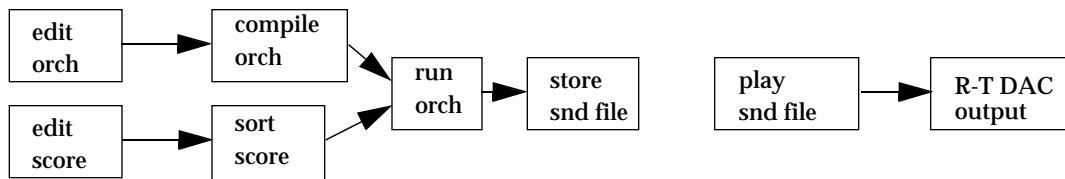


Figure 4. Steps of the SWSS "Compilation" Process

The actual interaction with the batch programs also varies widely among SWSS systems, and shows the generations of user interface technology since the 1960s, progressing from card decks to terminal edit-compile cycles, to window-based incremental interactive front-ends with graphical description of instrument definitions and score editing.

2.4. SWSS System Issues

Before the detailed presentation of SC, we will discuss several of the central design issues in programming languages for DSP and SWSS.

Oscillator Unit Generator Issues

Several types of oscillators are used in SWSS systems. Interpolating oscillators give higher fidelity but require more computation (See [Roads 1996]). Other systems allow

you to choose the size of the wave table to get higher fidelity. Good SWSS systems provide both options—interpolating oscillators and variable table length—so that the user can determine what the most important factors are. This is typical of the “speed vs. space” trade-offs often found in software engineering.

Audio and Control Rates

To save computation time (often an important issue in SWSS systems), some implementors use different sampling rates for audio signals and control signals such as envelopes. Unit generators, whether standard oscillators or not, may write their output values every sample, or they may be told to update less frequently, e.g., every 32 samples (to save computation). Some SWSS languages (including SC) allow flexible, run-time specification of control and audio rates and/or control and sample buffer sizes, though this provides neither a fully type-safe programming environment, nor a flexible model of multi-rate signal processing, and introduces myriad opportunities for aliasing problems and artifacts.

SWSS Usage

Among the reasons for the rekindling of interest in SWSS systems in the 1990s is the increasing computational power of inexpensive computers, which means that the issue of flexibility outweighs that of non-real-time performance. (In fact, SC can compute complex instruments in real-time on hardware costing less than US\$ 2000.) The flexibility of SWSS input languages and the continued popularity of the modular synthesizer model has meant that there is widespread interest in the construction of interfaces to real-time MIDI/DSP systems using this paradigm. Systems that merge features of SWSS and real-time performance systems are exemplified by (e.g.) *Accelerando* (Lent, Pinkston, and Silsbee 1989), *Kyma* (Scaletti 1989), the *IRCAM Musical Workstation* (Lindemann et al. 1991), and now SC.

Advanced User Interfaces for SWSS Systems

One of the immediate problems in building sophisticated scores is the complexity of specification of break-point time envelopes and of wave tables via Fourier overtone summation. Graphical editors for envelopes and overtone spectra emerged soon after appropriate graphical displays and input devices were developed, and now come in many flavors. Most synthesis languages, however, do not include any but the most rudimentary graphical tools, and the available shareware extensions often leave much to be desired. SC is an exception in this area as well.

Part 3. The SuperCollider Language

3.1. SuperCollider Language Syntax



SuperCollider is a modern object-oriented programming language; its syntax (how expressions are structured) is a mix of C++ and Smalltalk. If you know any modern programming language, most of the elements of SC will be familiar to you. One can program in two different styles in SC: *function-oriented* or *message-passing*; these will be described below.

In this Part, we will briefly introduce the basic elements of the language, and give some expression examples. It will certainly help if the reader has some programming background in another high-level procedural or object-oriented language. The extended examples in the next Part will use the language elements described here.

Types of Statements

As in most common programming languages, SC programs are made up of a sequence of statements; these typically correspond to lines of the program. A statement can be one of several types.

Comments are ignored by the compiler, and allow you to document your programs for easy reading.

Declarations are your way of introducing a new variable name to the compiler (e.g., “I’m going to use the name ‘osc’ to refer to my oscillator.”).

Assignments allow you to give value to a variable (as in “Set x to be 7.”).

Control structures allow you to control the flow of the execution of the program (e.g., “If the pitch is a C, then do this, otherwise, do that.”).

There are also many kinds of *message-send* statements that allow you to create and manipulate data objects. The following sections describe the various kinds of statements in more detail, and give examples of their usage.

Comments

Comments allow you to put descriptive text in your programs that explain their construction in plain English; single-line comments can be introduced by “--” (two hyphens, not an em-dash) and continue to the end of the line. Multi-line comments are enclosed with “(“ and “)”

Examples

```
-- This is a 1-line comment.
```

```
(* This is a multi-line comment.  
   ...  
)
```

Comments are *highly* recommended, as they make it much easier to read SC programs (help avoid write-only code). Comments should describe any program expressions that are not extremely obvious, and should give the useful ranges for variable values.

Statement Separators

SC expression statements are *separated* (rather than *terminated*) by semi-colons (i.e., as in Pascal rather than C). This means that all statements except the last one in a block are ended with “;” It is not an error, however, to have extra semi-colons in your programs, so you can safely end every statement with a semi-colon.

Several statements can be written on one line, if they are separated by semi-colons.

Examples

The following two examples are equivalent.

```
freq = 440;  
duration = 1.0;  
amplitude = 1.0;
```

and

```
freq = 440; duration = 1.0; amplitude = 1.0;
```

Data Scope and Extent

SC provides several kinds of variables that behave differently in terms of their scope (where is the name defined) and extent (how long does the variable live).

Normal Variables

Normal variables have local scope, i.e., they are only defined within the function in which they are declared. If they are declared outside of a function, they are defined for the entire file (this is called *global* data). Function arguments have the scope of the function they’re defined in.

Static Variables

There is only 1 copy per program of a variable that is declared as *static*. This declaration uses the word *static* in the place of the normal *var* declaration. (All global variables within a program are by nature static.)

Constants

A constant is a variable that cannot be changed after its declaration and initial assignment. To create a constant, use the `const` keyword and provide a value at declaration time, as in the example,

```
const fortissimo = 0.95;      -- Declare and assign a value to a constant
```

Arguments

Function arguments are declared after the “{” that starts the function (see the section on functions below). Their scope is the function in which they are declared.

Block-Format Data

SC data variables can be categorized into 4 groups depending on their temporal behavior.

Passive Data

Most data in a program is passive; it does not change until you assign a new value to it by placing its variable name on the left-hand side of an assignment.

Audio-Rate Variables

Audio unit generators such as oscillators create *audio-rate* objects. Sending the value message to one of these will generally return a sample buffer (an array of 64 samples by default). Built-in constructor functions that create audio-rate objects generally begin with “A” as in the `AsinOsc` in our introductory example that created and returned an audio-rate sine oscillator object. Note that it is the name of the constructor function that implies that this is an audio-rate variable; the name of the variable in which we hold the oscillator object is arbitrary (as in `osc` in the example). This is different from the usage in FORTRAN-style languages (e.g., Csound), where the variable name implies its type.

Control-Rate Variables

Variables that are to be used as control signals (e.g., envelopes), can be represented as control-rate objects. The constructor functions for these generally start with “K” as in the `Ktransient` control-rate envelope generator. These values change at the control rate (1/64th of the sample rate by default, though it can be changed).

Polled Variables

Polled variables change over time, but return a single value whenever they are read. These can be used, for example, for phrase-level variables that are read once per note (see the spatial panning example below).

Variable Declarations

As in most programming languages, one can name data items (variables); this follows the usual steps of *declaration* (defining what names are to be reserved for variables), *assignment* (giving values to variables), and *reference* (accessing the variable's value for use in an expression). To declare a name for a normal variable, one uses the `var` declaration. If you look back at the simple example from the introduction, you'll see that we declared the oscillator and output buffer variables with the statement,

```
var osc, outval;          -- Declare 2 variable names (no type information is provided).
```

We then used these names in assignments and in references in the output expressions.

Variable names in SC are untyped; one can assign different types of data to the same variable (though it is considered quite questionable style), as in the following example.

```
var data;                -- Declare the name "data".
data = 4;                -- Assign a number to data.
. . .
data = "hello";          -- Assign a string to data.
```

There are also type-specific declaration expressions for several special types of data; returning again to the initial example, to declare the names of our sound output buffers, we used the expression,

```
defaudioout L, R;        -- Define outputs named L and R.
```

The list below gives the various types of special declarations; each of the data types mentioned here will be described below.

<code>defaudioout</code>	Declare an audio output buffer.
<code>defaudioin</code>	Declare an audio input buffer.
<code>deftable</code>	Declare a wave table (or sampled envelope).
<code>defenvelope</code>	Declare a break-point envelope.
<code>defdelay</code>	Declare an audio delay line.
<code>defaudiobuf</code>	Declare an audio sample transfer buffer.
<code>defbus</code>	Declare a bus for passing sound between instruments.

Table 2: Special Typed Declarations

Optional Declarations

Variables can be explicitly declared (i.e., used in an assignment without being previously declared). This is generally discouraged as it can make programs more difficult to debug.

Names

User-defined variables and functions can be arbitrarily named, as long as the first character in the name is a letter. Variables may not have the same names as SCreserved words or built-in functions.

Examples

```
aValue, functionName    -- Two legal names
2Another, var, Asinosc   -- Three illegal names
```

Assignment

To provide a value for a variable, use *assignment*. The variable name is placed on the left-hand side of a “=” and some expression that returns a value is on the right, as in,

```
name = expression;      -- Assign the value of expression to name.
data = data + 1;         -- Increment the variable “data.”
```

One often pronounces “=” as “gets” so that one would read the last statement above as “data gets data plus one.”

The left-hand side of an assignment must be a variable name; it cannot be an operation expression, as in the example,

```
data + 1 = 7;           -- Illegal assignment
```

Return Value

If you want to return a value from a function, use the caret (^, typed as <SHIFT>-6), as in the example,

```
^ expression;
```

Storage Reclamation and Garbage Collection

Variables that are no longer needed (i.e., for which there are no more references in running code) are declared as “garbage” and are automatically freed (collected) by an incremental *garbage collection* (GC) process. The user interface message view’s header shows the percentage of time that the system spends going garbage collection.

3.2. SuperCollider Data Types

SC supports the common data types found in languages of the FORTRAN family (e.g., C) (integers, floating-point numbers, characters, strings, etc.) in addition to which it has several useful extensions taken from advanced languages such as Lisp and Smalltalk (e.g., lists and closures). These are summarized in the following sections.

Numbers

Number variables can represent integer (whole number) values (e.g., 3) or floating-point real numbers (e.g., 3.14126), and can be described in several formats (base-10, hexadecimal, etc.). Special numbers such as `Pi` and `infinity` are supported. By default, all integers are stored as 32-bit values, and all floating-point numbers in 64-bit “double-precision” format.

Numbers support the expected arithmetic, logarithmic, and trigonometric functions. There are also functions for manipulating complex numbers, for integer division, and for range mapping (e.g., soft clipping).

Examples

21, 0.443	-- Integer and floating-point numbers
0xff	-- Hexadecimal (base-16) for 255
21 + 4	-- Simple addition
3 / 4	-- Divide 3 by 4
8.rand	-- Answer a random integer between 0 and 8
	-- (i.e., send the message “rand” to the number 8).
2.5.rand2.post	-- Create a random floating-point value in the range +/-2.5
	-- with the message rand2 and print it to the message output
	-- view with the post message.

In this last example, the “.” character serves two purposes—decimal point and “dot” for message-passing. Think of rewriting the example `rand2(2.0).post` (see the discussion of function calls below). Note also that many messages (like `rand`) behave differently when sent to different kinds of objects (this is called *polymorphism*). The `rand` functions answer a number that is of the same type as their receiver); `8.rand` answers an integer, `8.0.rand` answers a floating-point number.

Strings

As in most other languages, text strings are represented as arrays of characters. In SC, strings are always written between double-quotes. Strings can be read from and written to files, as well as queried from the user and written to the message transcript view.

Examples

“This is a string.”

The maximum length of a string is 255 characters. The language has a wide variety of string functions such as file I/O (freadline, fwriteline), message view output (post), user prompting (getStringFromUser), and tokenizing (parse). One can also turn a string into a list of integers that represent the ASCII characters with the `spell` message; the `list2str` message does the opposite. Any variable can be printed as a string by sending it the message `asString`.

Symbols

A Symbol is a string that is unique, i.e., all instances of a symbol that's spelled the same are exactly the same object. Symbol values can be written using single quotes, or the symbol can be preceded by a backward slash, as shown in the examples below. Symbols are typically used for special system constants, e.g., `\nil`.

Examples

```
'Symbol'          -- Single quotes for a symbol.
\Symbol           -- Back-slash for a symbol.
```

Built-in functions allow you to convert between symbols and lists of integers (`list2sym`), and between symbols and references (`resolveName`).

Lists

SC lists are collections of objects that maintain their order; they can be used as indexed arrays of 1 or more dimensions, or as Lisp-style lists or Smalltalk-style collections. One can use a numerical index to get at a value in a list using the “@” message as in `list @ index` (noting that list indices are zero-based—the first item in the list has the index 0); “@@” is an “auto-wrap-around” indexing operator, as illustrated in the examples below.

One can also access lists as C-style linked lists or Smalltalk ordered collections. For example, you can insert elements in the middle of lists, or add/remove items at either end of the list.

All elements in a list do not have to be of the same type; in fact, one often mixes numbers, strings, and symbols in lists.

Examples

```
[1 2 3]           -- 1-dimensional list (like an array).
[[1 2][3 4]]      -- 2-dimensional list (list of lists).
[ 1 \hello]        -- You can mix types in a list.
[3 4 5] @ 1        -- Answers 4 (!) (the second element).
[3 4 5] @@ 6       -- Answers 3 (wrapping around).
```



```

[1 2 3 4].put(2, 9)      -- Put an item in the middle of a list (number 9 at index 2).
                        -- Answers [1 2 9 4].
[1 2 3 4].insertAt(2, 9) -- Insert an item in the middle of a list; answers [1 2 9 3 4].
[a b] $ [c d]           -- Concatenate lists; answers [a b c d].
[6 2 0].max             -- Answer the maximum value of a list (6).
[1 2 3].mirror          -- Answers [1 2 3 2 1].
[4 2 8].sort            -- Answers [2 4 8].
[L R].choose            -- Choose an element at random.

```

There are many more built-in functions and interesting control structures made using lists, as we will illustrate in the following chapters.

Closures

A closure (or *block* or *continuation*) is just like a function, but it has no name; closures can be stored in named variables, or passed to functions as arguments. Like a function, a closure can take zero or more arguments, and can be evaluated as many times as you like (by sending them the message value with or without arguments). In SC, closures are written between curly braces (`{...}`).

As an example (manual p. 43), we can construct a numerical counter. The counter is implemented as a closure that increments (and returns) its value each time you evaluate it. In the code example below, we create the counter (closure) and hold it in a variable named `counter`. Sending the message value to the closure evaluates it, incrementing and returning the counter's value.

```

var current, counter, x;      -- Declare names for the counter value and closure.
current = 0;                 -- Set the starting the counter value.

counter = {
  current = current + 1;      -- make a closure (between the curly braces).
  ^current;                  -- First increment the counter's value.
};                           -- Then answer (return) the number.
                             -- End of the closure.

counter.value.post;          -- To use this, execute the following.
x = counter.value;           -- Evaluate the closure; print 1 to the message view.
                             -- Evaluate the closure; x gets 2.

```

Code Example 10. Example of the Use of Closures

In SC, all top-level functions are named, and all functions defined within another function are treated as closures (i.e., there are no nested function definitions as in Pascal).

References

A reference is used to give the name of an item that is not a normal data variable, e.g., a function. References are denoted with back-quotes (``fcn_name``) and can only be sent certain messages. One can convert a symbol into a reference using the message `resolve-Name`.

Example

```
`fcn_name          -- A reference to a function name.
```

Special Variables

There are several pre-defined variables in SC; these are given in the following table.

<code>now</code>	The current clock value.
<code>sr</code>	The sampling rate (default 44100).
<code>frameSize</code>	The frame size (default 4096).
<code>subFrameSize</code>	The sub-frame size (default 64).
<code>thisFunc</code>	A reference to the current function.
<code>this</code>	The receiver object (see OOP below).
<code>super</code>	The super-class instance (see OOP below).
<code>mouseX, mouseY</code>	The x- and y-coordinates of the mouse
<code>\nil</code>	The symbol meaning “nothing” or “uninitialized.”
<code>\end</code>	The symbol meaning “end of stream.”

Table 3: Special Variables in SC

Vector Signal Data

There are several types of data arrays that are treated specially in SC.

Table

Wave table functions are declared using the `deftable` keyword seen above. Once defined, these tables can be edited using the SC user interface’s table editors (introduced below). You can, for example, create table data using Fourier summation of sine waves, frequency modulation (FM), or random-valued noise.

Envelope

Envelopes are generally used for control-rate functions that are described using break points (corners of the envelope function) between which data is interpolated. The are declared using `defenvelope` and can be edited by hand by moving the break points with the mouse. The editor is introduced in the chapter on the SC user interface.

Signal

A signal or audio buffer is simply a block of samples (AKA subframe buffer). By default, these are 64 samples in length. (This size can be set using the **Set Globals** dialog described below.) Audio buffers can be declared using the `defaudioin` and `defaudioout` expressions. They are read and written using the `in` and `out` messages, respectively.

Examples

```
defaudioin left;           -- Declare an audio inout buffer.
in(left).clip.distort      -- Read the audio input, then clip and distort it.
```

Delay Line

A delay line is a sample FIFO (first-in first-out) with variable reading position. These are declared using the `defdelay` expression and read using the `tap` message. When you declare a delay line, you set its maximum length; you can then read it with one or more taps, and the taps can move within the delay line's length.

Bus

Busses can be used to pass signals between instruments, or for multi-stage processing within an instrument. A bus is declared using the `defbus` declaration, and read/written with `in` and `out`.

3.3. The Syntax of Function Calls and Messages

Functions in SC can be called using C-style “applicative” syntax (apply a function to a variable), as in,

```
x = sin(y)                -- Function call; function name comes first followed by the
                           -- argument names in parentheses.
```

or using C++-style “dot notation” (send a message to an object), as in,

```
x = y.sin                 -- Message-passing; message receiver comes first.
```

In FORTRAN-style languages (e.g., C), function calls are “stand-alone,” and take arguments in parentheses after the function name (e.g., `sin(y)` to take the sine of `y`). In pure message-passing languages (e.g., Smalltalk), messages are always sent to some specific receiver object (e.g., `x sin` to send the message `sin` to the object in the variable `x`). In Smalltalk, the message and the receiver are separated by white space (as in `x sin`); in C++ (and SC), they are linked together with a dot “.” meaning “send” as in `x.sin`.

In SC, the function-calling and the message-passing forms are equivalent, so that `sin(x)` and `x.sin` have the same effect.

The first argument of a function (in C syntax) can be treated as the message receiver (in Smalltalk syntax)—e.g.,

```
f(a, b) = a.f(b).
```

To send the message value to an oscillator (getting its output value) and then send this value to the output channels, one could write,

```
val = value(osc);      -- get the value
out(val, L);           -- send it to the left channel
out(val, R);           -- send it to the right channel
```

or, more tersely,

```
value(osc).out(L).out(R);
```

or,

```
osc.value.out(L).out(R);
```

or even,

```
out(out(value(osc), L), R);
```

All of these forms of the expression are equivalent.

Function composition ($g(f(x))$) can then be expressed as *cascaded* message-sends such as `(x.f.g)`, which means “send the message `f` to object `x`, and send the message `g` to the result of that.” In the example we presented above, you could rewrite,

```
2.5.rand2.post          -- Cascaded message-sends (Smalltalk style)
```

as,

```
post(rand2(2.5))        -- Composed function calls (C-style)
```

The “default” message for any object is `value`, so that one can write `x.` to mean `x.value`, as in,

```
{ (osc. * env.).out(chan) }.dspAdd;    -- The same as saying osc.value...
```

Function Definition

A new function can be defined by giving it a name, an argument list, optional temporary variables, and the statements that make up its body, as in the following example, which defines a function that returns the sum of its two numerical arguments.

```
-- Function that answers the sum of its two arguments
--
summer {                                -- The name of the function is "summer."
  arg a, b;                            -- Here are the arguments (declaration required).
  var c;                               -- It has a local variable (declaration optional).

  c = a + b;                           -- Function body, assign the sum of a and b to c.
  ^c                                   -- There's a return statement (optional).
}                                       -- End of the function.
```

Code Example 11. Example of Function Definition

Functions can be called before they are defined (i.e., you can use a function name in an SC program before you've come to that function's definition).

Function Names

SC functions that generate audio-rate signals have names that start with "A" (e.g., *Aformanta*); the names of those that generate control-rate signals start with "K" (e.g., *Ktransient*). Polled functions start with "P" (e.g., *Psinosc*). User-defined functions are free to violate this, however.

3.4. Control Structures

Any programming language must provide the programmer with a means to control the flow of execution within a program. One might want, for example, to have the timbre change for alternate notes, so one would assign different values to some parameters depending on the state of a counter. The language expressions that allow one to do this are called *control structures*.

Standard C/Pascal Control Structures

Common control structures are *branching* ("if it's raining, take an umbrella, otherwise not."), *looping* ("add up the totals for the 12 months of last year."), and *switches* or *case statements* ("if fish, chardonnay; if beef, cabernet; if pasta, chianti."). SC includes these standard control structures, as well as a number of more advanced ones borrowed from Smalltalk.

Program Redirection

The simplest (and most unwise) control structure is the goto statement. One can give a name to an arbitrary program statement with the label expression, and then jump there from somewhere else with the goto statement.

```
label name;  
...  
goto name;
```

Goto is considered unwise because it makes the program flow difficult to trace.

Branching

Simple branching can be controlled by conditions using the if/then/else/end.if structure. As an example, if one wanted to have louder notes have a larger modulation index in an FM instrument, one could use the following expressions.

```
if (amplitude > 0.75) then  
    index = index * 1.2;  
end.if;
```

The first expression in this statement is the condition (amplitude > 0.75). It returns a Boolean value (i.e., true or false). The condition statement does not have to be enclosed in parentheses, but I find that it improves program readability. The condition statements can be complex (i.e., if ((x > 0) && (y > 0)) for the case that both x and y are strictly positive). There can then be as many statements as needed in the then...end group.

The else clause can be used to make a two-way branch structure, as in the following example that places notes on the left or right channel depending on their pitch.

```
if (pitch > 48) then  
    position = -1;  
else  
    position = 1;  
end.if;
```

The else clause can have a condition of its own using the elseif keyword, as in,

```
if (pitch > 48) then    position = -1;  
elseif (pitch > 24) then position = 0;  
else                  position = 1;  
end.if;
```

Looping

SC has both *conditional* loops (do this as long as (or until) the condition is true) and *iterative* (for) loops. The conditional loops generally involve a test expression (the condition) that answers a Boolean value, and a group of 1 or more expressions.

```
while (some Boolean expression) do
  statements
end.while;
```

The for loop typically uses a *start* expression (done once at the start of the loop), a *condition* (tested as each iteration), and a *step* expression (executed between iterations), as in the following example, which will execute the given statements 100 times with the variable named *i* (to which we can refer in the statements) taking on successive values from 0 to 99.

```
for i = 0; i < 100; i = i + 1; do
  statements
end.for
```

The start expression is executed once at the start of the loop; in the example above, it sets the loop counter *i* to zero. the condition compares *i* with 100, ending the loop when it reaches that value, and the step expression increments *i* by one.

Case Statements

One can have a set of options that depend on the state of a variable by using *switches* also known as *case statements*. The case expression can be any atomic value (i.e., a symbol or number). In the example below, we assign a note's position based on the name of the instrument that plays it.

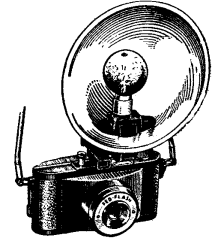
```
switch instrument      -- Set the position depending on the instrument.
case 'fm_instr         -- The instrument name is a reference.
  position = 1.0;      -- Set the position value.
  break;              -- Exit from the switch if this case is true.
case 'grain_instr
  position = 0.4;
  break;
case 'noise_instr
  position = -0.6;
  break;
default                -- This for all cases that are not handled by the above clauses.
  position = 0.0;
end.switch;
```


Smalltalk List Control Structures

Smalltalk and other advanced languages have more powerful and flexible control structures than those introduced above. Most of these involve messages sent to lists or closures.

Repetition

The `timesRepeat({})` message is sent to a number (probably an integer) and takes a closure as its argument. It repeats the argument as many times as the receiver number denotes.



```
number.timesRepeat ({ some closure to be repeated number times });
```

Note the syntax of this expression; because the argument of `timesRepeat()` is a closure, we write the expression as `number.timesRepeat({ statements });` this form (closure-as-argument) will be used for the list control structures below. In the case of `timesRepeat()`, the closure must be one that takes exactly one argument (which will be assigned to the integers between 0 and `number - 1` for the iterations through the loop).

List Iteration

There are a number of control structures that are used as messages sent to a list, and allow you for example to test the items in a list, to select those items that fulfill a given condition (expressed in a closure), or to apply a closure to all the items in a list.

The simplest list iterator is `forEach()`, which applies a given closure (the argument of the message) to each item in the list to which the `forEach()` message is sent. The argument of `forEach()` is a closure that takes two arguments: the element's value and the element's index. This closure will be called for each item in the receiver list, as in the following example, which prints the items in the receiver list to the transcript view 1-per-line.

```
[1 2 3 4 5].forEach({ arg item, index; item.post});-- Prints list to message view
-- 1 item per line.
```

The `collect()` message applies its argument (a closure that returns a new item) to all of the items in the list to which it is sent; it answers a new list of the results of that application. For example, to square the items in a numerical list, use the following.

```
[1 2 3 4 5].collect({ arg item, index; ^item * item});-- Answers [1 4 9 16 25].
```

The `collect()` message can also be used to select items from a list that fulfill a given condition. If you return the special symbol `omit` from an iteration through the closure, then

the corresponding element is not added to the result list. For example, to select the elements from a numerical list that are between 60 and 72 (inclusive), you could use the following statement.

```
[ 58 66 84 73 61 65 80 63 ].collect(
  { arg item, index;
    if ((item >= 60) && (item <= 72)) then
      ^item
    else
      ^\omit
    end.if}) -- Answers [ 66 61 65 63 ].
```

To locate an item in a list that fulfills a condition, use `find()`; it answers the index of the first item it finds for which the given closure is true. For example, to find the first even number in a list, use,

```
[1 3 5 7 8 9 10].find({ arg item; ^item.even}) -- Answers 4 (index of 8 in the list).
```

The functions `any()` and `every()` test the items in a list using a user-supplied closure and answer whether the closure is true for any or all, respectively, of the list's items.

There are many other interesting list control structures; some of them will appear in the SC program examples that follow.

3.5. Built-in Functions

SC is a relatively full-featured language, there is a comprehensive library of basic functions (methods) for mathematical operations, list processing, sound synthesis and DSP, file and screen I/O, and musical tasks. I will outline them below, and demonstrate them in the examples that follow.

Mathematical and Numerical functions

SC supports basic arithmetic, Boolean logic, bit operations, logarithmic and exponential functions, trigonometry, rounding and truncation, range mapping, folding, and wrapping, polynomials, sign, primes, and coercion. In most cases, the “standard” C-style infix operators are used. The library also includes a wide range of random number generators, including Gaussian, bilinear, gamma, bilateral exponential, and Poisson distributions.

<code>+</code> <code>-</code> <code>*</code> <code>/</code>	Basic arithmetic
<code>+</code> <code>!</code> <code>-</code> <code>!</code> <code>*</code> <code>!</code>	In-place signal addition, subtraction, and multiplication
<code>%</code> <code>//</code> <code>**</code>	Modulus, integer division, “raise-to-the-power-of”
<code>min:</code> <code>max:</code>	Minimum, maximum

lcm: gcd:	Least common multiple, greatest common divisor
abs: neg: sign:	Absolute value, negation, sign
rand, rand2	Random number 0-to-x (rand) or -x-to-x (rand2)
coin, expiran	Coin toss, exponential random distribution

Table 4: Basic Numerical Methods*Musical Parameters*

There are also built-in functions for handling musical pitch and frequency, and dynamic amplitude or loudness values.

midicps/cpsmidi	Convert between MIDI key number and Hz.
octcps/cpsoct	Convert oct.pitch (floating-point octaves) to/from Hz.
dbamp/ampdb	Convert deciBels to/from 0-1 floating-point amplitudes.

Table 5: Pitch and Amplitude Conversion Functions*List-Processing*

SC borrows Smalltalk’s ordered collection operations for 1- or N-dimensional lists. Lists can be treated as arrays (indexed by integers), dictionaries (indexed by “objects”), or sets (not indexable), and they support a rich variety of functions for element testing, copying, sorting, arithmetic, etc. There are also Smalltalk -style list iteration methods, such as `collect`, `find`, and `forEach`. Lists can even behave like Smalltalk dictionaries, where items are associated with “keys” (usually symbols) and can be looked up using their keys. The most common list functions are given in the table below (*i* is used for numerical list indices, *x* for arbitrary values, and *func* for function closures).

Basic List Access

@	Index into a list (zero-based).
@@	“Wrap-around” indexing (go back to the start after final index).
[@]	“Clipped” indexing (repeat the last value).
@ @	“Folded” indexing (read through the list in retrograde after the end).
size	Answer the size of the list.
\$	Concatenate two lists.
#!	Append one list to another.
choose	Get a random value from the list.

Adding and Removing Items

add	Add values at the end of list.
addFirst	Add value at the start of list.
put(i, x)	Put <i>x</i> at the position given by <i>i</i> .
insertAt(i, x)	Insert <i>x</i> at index <i>i</i> (shift the rest right).

<code>first(n)</code>	Get the first n values (default $n = 1$).
<code>last(n)</code>	Get the last n values (default $n = 1$).
<code>take(i)</code>	Remove and answer the item at index i .
<code>swap(i, j)</code>	Swap the elements at indices i and j .
<code>reverse</code>	Reverse a list in-place.
<code>scramble</code>	Randomly re-order the items in a list.
<code>rotate(n)</code>	Rotate the list n places.
<code>permute(n)</code>	Answer the n th permutation of the list.
<code>stutter(n)</code>	Repeat the elements in the list n times.

List Hierarchy Processing

<code>flat</code>	Answer a 1-D (i.e., flat) list from a (possibly) multi-D list.
<code>clump(size)</code>	Divide the list into sub-lists of size <code>size</code> .
<code>curdle(prob)</code>	divide the list into sub-lists with a probability of <code>prob</code> of breaking between any two elements.
<code>flop</code>	Swap rows and columns of a 2-D list.

List Element Testing

<code>indexOf(x)</code>	Get the index of item x or answer <code>\nil</code> if it's missing.
<code>includes(x)</code>	Answer whether the list includes the item x .

List Control Structures

<code>any(func)</code>	Answer whether the function is true for any items in the list.
<code>collect(func)</code>	Answer a list with the result of applying the function to all items.
<code>forEach(func)</code>	Apply the function to each item in the list.
<code>find(func)</code>	Answer the first item for which the function is true.

List Math

<code>min/max</code>	Answer the minimum or maximum values in a list.
<code>integral</code>	Answer the sum of the items in a list.
<code>sort</code>	Sort a list into numerically ascending order.

Lists and Strings

<code>list2str</code>	Take a list of ASCII integers and answer a string.
<code>list2sym</code>	Take a list of ASCII integers and answer a symbol.
<code>spell</code>	Convert a symbol or string to a list of ASCII integers.

Lists asSets

<code>asSet</code>	Remove duplicate items from the list
<code>union(list2)</code>	Answer the union of all the elements in two lists.
<code>sect(list2)</code>	Answer only the items that are in both lists (the intersection).
<code>removeAll(list2)</code>	Remove all the items in <code>list2</code> from the receiver list.

List Data Creation

ramp(size, start, step)	Create a linear ramp function.
white(size, lo, hi)	Create a list with white noise values.
pink(size, lo, hi)	Create a list with pink noise (low-pass filtered) values.
brown(size, lo, hi, step)	Create a list with a brownian walk.

Example

```
brown(8 36 72 12).curdle(0.4).stutter(3).scramble.flat;
```

Table 6: Common List Functions*Sound Synthesis*

There are a wealth of signal sources in the language, including sine, cosine, and wave table look-up, formant, FM, PM, wave shaping, granular, vector, and band-limited pulse train oscillators. Various types of noise generators are included, along with sound file and live audio input functions. The table below gives the most common signal synthesis and control functions. The (AKP) before the function type denotes whether audio-rate, control-rate or polled versions of them exist. The (i) at the end means that an interpolating version of the unit generator exists. An (a) means there is a version with low-frequency amplitude modulation (tremolo).

Generators

(AKP)oscil(ia)(table, frq, phase)	Wave table oscillator.
(AKP)sinosc(i)(frq, phase)	Sinusoidal oscillator.
(AKP)coscil(i)(table, frq, bFrq)	Chorusing oscillator (dual oscillators with slightly different frequencies leading to beats).
Aposcil(cFrq, mFrq, index)	FM oscillator pair.
Avoscili(tableList, frq, index)	Multi-table vector oscillator.
Apulse(table, frq, formFrq)	Table pulse oscillator with delay.
Aformant(frq, formFrq, bw)	Formant oscillator.
Acpgrain(buf, offset, rate, dur, amp, complFunc)	Parabolic grain generator.
(AK)noise(frq, amp)	Noise generator (comes in many flavors).

Filters (these take a signal input as an argument to the value message in the DSP loop)

Atone(frq)	1-pole low-pass filter.
Alpf(frq)	2-pole low-pass filter.
Arlpf(frq, Q)	2-pole resonant low-pass filter.
Aatone(freq)	1-pole high-pass filter.
Ahpf(freq)	2-pole high-pass filter.
Arhpf(freq, Q)	2-pole resonant high-pass filter.
Abpf(frq, bw)	Band-pass filter.
Abrf(frq, bw)	Band-reject filter.

Envelope Generators

(KP)const(value, dur, complFunc)	Finite-duration constant value.
(AKP)line(start, stop, dur, complFunc)	Line segment generator.
(AKP)xline(start, stop, dur, curve, complFunc)	Exponential line segment.
(AKP)transient(table, dur, ampo, bias, complFunc)	Table-based envelope.
(AK)bpenv(env, scale, bias, time, complFunc)	Break-point envelope.
Atrienv(dur, amp, complFunc)	Triangle envelope generator.

Table 7: Common Signal Synthesis and Control Functions*Signal Processing*

In addition to filters, modulators, delay lines, and mixers, SC supports several kinds of distortion, clipping, windowing, gates, dynamic range processing, and other DSP operations. There are relatively few audio analysis functions (e.g., pitch detection), however.

Delay Lines

Adelay(buf, time)	Simple delay line.
Aallpassdly(buf, time, decay)	All-pass delay line.
Acombdly(buf, time, decay)	Comb (feed-back) delay line.
tap(i)(dLine, time)	Delay line tap.

I/O

out(buf)	Output writer.
in(audioln)	Input reader.
mixout(amp, output)	Multiplying output writer.
pan2out(pos, out1, out2)	Stereo output panner (pos is +- 1).
Abufrd(buf, offset, rate)	Audio buffer reader.
Abufwr(buf, offset)	Audio buffer writer.
Arecord(buf, complFunc)	Input-to-buffer recorder.

DSP

(AK)gate(in, trigger)	Audio gate.
(AK)latch(in, trigger)	Sample and hold.
(AK)lag(val, time)	Exponential lag function.
xfade(in1, in2, pos)	Cross-fade between two inputs (pos is +- 1).
xfadeEnv(in1, in2, pos)	Cross-fade between two envelopes (pos is +- 1).

Spectral Processing

shaper(table, index)	Wave shaper.
flip	Spectral interter (ring mod. by the Nyquist freq.)

Control

dspAdd(stage)	Add a closure to the DSP loop in the given stage.
dspStart(startFunc)	Start the DSP engine with the given function).
dspKill(flag)	Stop the DSP engine if the flag is true.
dspRemove	Remove the current task from the DSP loop.
sr, frameSize, subFrameSize	Answer the sample rate, frame size, or sub-frame size.

Table 8: Common Signal Processing Functions*I/O*

SC allows programs to read data from files or the keyboard, and to write messages to files or to the GUI's message view. There are simple functions for getting strings from the user using pop-up dialog boxes.

fopen(name, rw)	Open a file for reading ("r"), writing ("w"), or both ("rw"); answer a file identifier (a number).
fclose(id)	Close the file with the given id.
freadline(id)	Read/write a line of text from/to the file id.
fwriteline(id)	Answer a string.
freadlist(id)	Read/write a list from/to the file id.
fwritelist(id)	Answer a list.
parse(str)	Break a string into a list of tokens based on white space.

Table 9: File I/O Functions*GUI Interaction*

The most commonly used functions for interacting with SC GUI items are listed in the table below.

getItemValue(i)	Get the value of the GUI item <i>i</i> .
getItemValue2(i)	Get the secondary value of a range slider.
getItemString(i)	Get the string value of the GUI item <i>i</i> .
getItemList(i)	Get the list box list of the GUI item <i>i</i> .
setItemValue(i, x)	Set the value of GUI item <i>i</i> to <i>x</i> .
setItemValue2(i, x)	Set the secondary value of GUI item <i>i</i> to <i>x</i> .
setItemString(i, x)	Set the string value of GUI item <i>i</i> to <i>x</i> .
getStringFromUser(default, prompt)	Prompt and answer a string from the user.
mouseButton	Answer whether the mouse button is down.
mouseX/Y	Answer the x or y coordinate of the mouse.
addMenuCommand(item, func)	Add an entry to the user menu with name <i>item</i> calling function <i>func</i> .

Table 10: GUI I/O Functions

Musical Magnitudes: Pitch and Frequency, Loudness and Amplitude

There are models of pitch in Hz, MIDI key numbers, and decimal octaves and functions for translating between them. Amplitude can be represented as decibels, or absolute values. Metronomes are provided for describing beat-oriented tempo curves.

MIDI I/O and Conversion

MIDI input can be treated as control-only, or as event triggers. the functions listed in the table below can be used in instruments to poll MIDI controller values. Special *voicer* objects exist to map MIDI note-on commands to instrument invocations.

<code>ctrlin(num, chan)</code>	Answer the value (0 - 127) of MIDI controller num on channel chan.
<code>bendin(chan)</code>	Answer the MIDI pitch bend value (-8192 - 8191) on channel chan.
<code>touchin(chan)</code>	Answer the value (0 - 127) of the MIDI aftertouch on channel chan.

Table 11: MIDI Continuous Control Polling Messages

Additional versions of these calls exist that can map the MIDI control value ranges to arbitrary floating-point numerical ranges.

Miscellaneous Functions

There are a number of useful functions that do fit neatly into the above categories. They are listed below for our reference.

<code>type</code>	Answer a symbol denoting the type of a variable.
<code>isNumber</code>	Answer whether an item is a number.
<code>isAtom</code>	Answer whether an item is atomic (i.e., not a list).
<code>isNumeric</code>	Answer whether a list contains only numbers.
<code>isAtom</code>	Answer whether a list contains only numbers and symbols (i.e., no sub-lists).
<code>asString</code>	Answer a string that represents the variable.
<code>copy</code>	Answer a copy of a variable.
<code>post</code>	Print a number, list, or string to the message view.
<code>dump(depth)</code>	Print any object to the message view to the given depth of its structure (i.e., descend depth levels in the structure).
<code>resolveName</code>	Get a reference from a symbol (i.e., find the function named x).
<code>value(args)</code>	Evaluate a function with optional arguments.

Table 12: Useful Miscellaneous Functions

3.6. Other Language Features

SC has a number of language elements that, while sometimes convenient, are also dangerous, and tend to lead to a rather unreadable or difficult-to-debug programming style (“write-only programming”). I will outline these below, and generally discourage their use by novices (especially in light of the lack of a debugger in the SC environment).

Multiple Assignment

One can assign a list to several variables in one statement using the “#” before the assignment, as in the following example.

```
# a b c = [ 1, 2, 3 ];           (i.e., a = 1; b = 2; c = 3)
```

The number of variables to the left of the “=” must be the same as the number of elements in the list.

Ellipsis Assignment

A refinement of multiple assignment uses the ellipsis (...) before the last variable name on the left of the “=”.

```
# a b ... c = [ 1, 2, 3, 4, 5 ];   (a = 1; b = 2, c = [345])
```

Implicit Declarations

As mentioned above (and already discouraged), variables can be used (assigned into) without being declared.

```
c = a + b;                       (c not declared previously)
```

Optional Commas

Commas can be used to separate function arguments, variable declarations, or list items. There are a few cases where they’re required, but they can generally be left out.

```
[ 1 2 3 4 ] == [ 1, 2, 3, 4 ]
```

Variable Numbers of Arguments

All functions may be called with fewer or more arguments than they expect. Default values can be given in the argument declaration (arg a = 1;), otherwise, `nil` is the assumed argument. Extra arguments are ignored (see the next example).

Functions with No Arguments

These need no parentheses, for example,

```
-- Define a function that returns the sum of its two arguments
-- (but provides default values should the arguments be missing).

sum_function { arg a = 1, b = 2; ^a+b }

-- Examples of calling sum_function with and without arguments

z = sum_function(8, 2);      -- z gets 10.
z = sum_function(4);         -- z gets 6 (use the default for the second argument).
z = sum_function;            -- z gets 3 (use the defaults for both arguments).
```

Defaulted Arguments

Using “\” as an argument means to skip it and use the default or previous value, e.g.,

```
-- Create a formant oscillator. Aformanta takes arguments
-- (frequency, formant frequency, bandwidth, amplitude).
-- (Amplitude is left out at creation time.)
cosc = Aformanta(fn, 500.0.expran, 9.0.rand * 22);

-- Create an envelope function with function table 3
env = Ktransient(tbl3, 2.0, 0.1, 0, `dspRemove);

-- Add the formant oscillator to the DSP loop with the envelope
-- as its amplitude (leave the first 3 arguments unchanged).
-- (The arguments of the value() message are the same as the
-- constructor message Aformanta().)
{ cosc.value(\, \, \, env.value).out(output) }.dspAdd;
```

The DSP Cycle

SuperCollider has different control and audio rates. One differentiates between the sample rate and the control rate. Unfortunately, the manual uses the term *sub-frame rate* to mean the control rate. (The term *frame rate* is used to refer to the output sample buffer size.) The control rate (actually, the control buffer size) can be set using the **Set Globals** item under the **Synth** menu; the default is 64 samples per control frame.

There are four stages to the DSP engine, and one can have data passed between instruments by placing them in different stages. For example, sound-generating instruments might be placed in stage 0 and the reverberator instrument in stage 3. (I’ll give examples of this below.)

To control the signal processing, a user program explicitly adds and removes functions from the DSP queue of a specific stage using the `dspAdd` and `dspRemove` calls.

3.7. The Parts of a SuperCollider Program

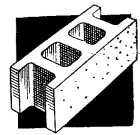
There are several parts to a typical SC program:

- Header—title comment, date, version, copyright, ...
- Declarations—declare output buffers, sound files, function tables, etc. (required)
- Init Function—run at compile-time, if present (optional)
- Start function—called at run-time if present; runs the instruments (normally present, though optional)
- Instrument functions—can be called from the start function

We will illustrate the use of each of these in the examples in the following sections.

3.8. Unit Generators as Objects

SuperCollider unit generators are classical objects (see the chapter on object-oriented programming in Part 6 if you're impatient) in that they have *constructor* methods and an *evaluation* method. As in C++, the class name of the unit generator is its constructor; `value()` is the default evaluation message, and generally (though not always) takes the same arguments as the constructor. The constructor is generally called once per note, and the evaluator is put in the DSP loop for continuous evaluation at run time, as in the following example (taken from the example instrument presented in the introduction).



```
var osc;                -- Declare a variable name for the oscillator.

osc = Asinosc(220, 0);  -- Unit generator constructor method -- name = UG type.
                        -- "Asinosc" arguments are (frequency, phase)
                        -- Create "osc" as an object (once per note).

{
    -- In the DSP loop (a closure evaluated once per control
    -- period), get osc object's value (a sample buffer), and send
    -- it to the two outputs in succession.
    osc.value.out(L).out(R);
}.dspAdd;               -- Add the above closure to the DSP engine's queue.
```

Code Example 12. Unit Generator Construction and Use of the `value` Message

To connect a control function (rather than a constant) to a unit generator, do so with an argument to the `value()` message, as in an instrument that uses frequency modulation

with an envelope for the modulation index. Because the index is controlled at control rate, we do not set it in the oscillator's constructor, but rather "plug it in" inside the DSP loop, as shown in the following example.

```

-- Create an FM Oscillator. The constructor's arguments are
-- (carrier_freq, mod_freq, mod_index).

osc = Aposcil(freq, freq*ratio, 0);    -- (Ths modulation index is 0 for now.)

-- Create an amplitude envelope.
a_env = Ktransient(env1, 1, 0.8, 0, `dspRemove);

-- Create the modulation index envelope.
i_env = Ktransient(env2, 1, 1, 0, `dspRemove);

-- DSP loop (closure).
-- This call tovalue() leaves the first 2 arguments alone but
-- plugs in the mod_index as the third argument.

{ (osc.value(\, \, i_env.value) *! a_env.value).out(L).out(R) }.dspAdd;
```

Code Example 13. Connecting Unit Generators inside the DSP Loop

3.9. SuperCollider Style

There are several elements to coding style: naming conventions, code documentation, program indentation are perhaps the most important.

Variable and Function Naming

If you want other people to be able to read your programs (or to be able to reverse engineer them yourself six month hence), it is important to use variable and function names that will help the reader understand your intention. The natural trade-off here is between terseness (the famous C variable *i*) and verbosity (my most famous Smalltalk class name *NoviceNavigatorSwissArmyKnifeController*). Function names should generally give a hint as to what the function does, or what it answers.

In Smalltalk, we recommend that important variable names tell the reader both the role and the type of the variable; for example, a geometric point that's used as the extend of a rectangle might be called *extentPoint*. A buffer used for sound output might be named *outputBuffer*.

There are two styles for naming variables in SC: C++ style and Smalltalk style. In C++, programmers often use embedded underscore "_" to separate the components of a complex variable or function name (e.g., *output_buffer* or *read_score*); in Smalltalk, we

use embedded upper-case letters for this (`outputBuffer` and `readScore`). The naming of the built-in library functions and the SC manual are a bit inconsistent, but use Smalltalk style more frequently (e.g., `forEach`, `timesRepeat`); in this book I generally use Smalltalk-style.

SuperCollider is case-sensitive. Variables named `osc` and `Osc` are different.

The names of the built-in functions and control structures are reserved, and cannot be used for variables or added functions. There is an exact list of these in the SC manual.

Program Documentation

Comments make a program easier to read. They also provide an important debugging tool. If you state your intentions in English before writing the code, you have a fully redundant human-readable specification against which you can check the implementation. As illustrated in the examples I present below, I believe that there should always be more English than program code on a page (I actually hold to this rule most of the time in my professional Smalltalk programming).

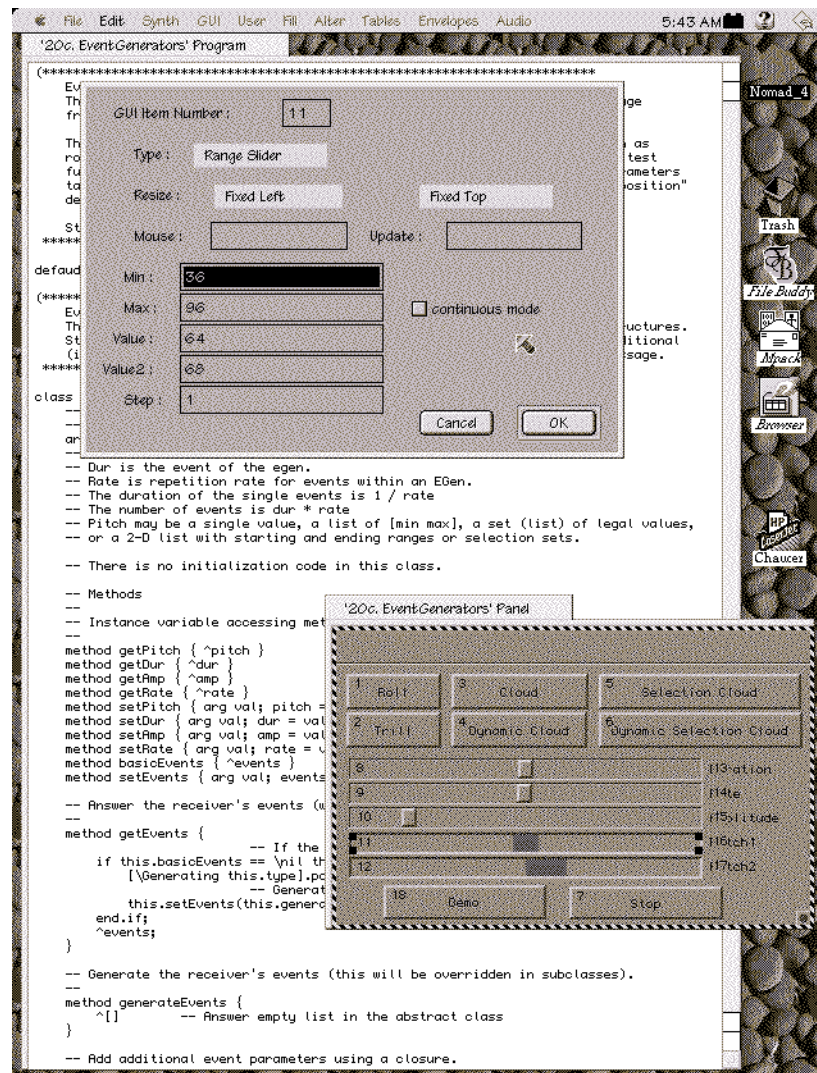
Code Formatting

Programmers generally use indentation to make the program flow visually obvious. As is evident in the code examples above, function definitions are indented one level (one tab or four spaces) relative to the function header. Control structures (`if/then/else`, loops, etc.) are frequently used as landmarks in a program, and are indented one level.

I prefer an indentation style called “two-column coding” whereby verbose comments are indented well to the right, so that one can easily read just the comments as a separate stream of text, with the executable code as the “left column.” This is the style I have been using in the examples in this book.

Program Organization

As presented above, good style also entails good program organization, with a header describing the program, global variable declarations, `init()` and `start()` functions, followed by user-defined function methods. You’ll see many examples of this in the chapters that follow.



Part 4. The SuperCollider User Interface

This Part will introduce the basic elements of the SC user interface and outline their uses.

4.1. SuperCollider Windows

On start-up, SC presents you with three windows: a program text editor, the SC message output view, and an instrument user interface view.

Program Text

The main SC program text view is a standard text editor. Only one file can be at a time, though you can copy a text from one file, then open another and paste the text. There are editor menu items for the typical text editing operations such as cut/copy/paste, as well as programming-related operations such as adding special delimiters, and a user-definable menu. An example of the text window is shown in the figure on the right.

```
-- OSC, FM Score Program
--
-- FM osc. with score
-- stp@create.ucsb.edu -- 1997.02.27

defauidout L, R;
deftable tabl1, env1, env2;

start {
  -- The score calls the instrument.
  [0, 'fm_instr', 1, 36, 2, 1.02].sched;
  [0.5, 'fm_instr', 1, 40, 2, 1.04].sched;
  [1, 'fm_instr', 1, 43, 2, 1.02].sched;
  [1.5, 'fm_instr', 1, 48, 2, 1.04].sched;
}

fm_instr {
  -- Simple FM
  arg duration, freq, index, ratio;
  var osc, i_env, a_env, hz;

  hz = (freq + 12).midiaps;

  -- args (carrier_freq, mod_freq, index)
  osc = Aposcil(hz, hz*ratio, 0);

  -- args (table, dur, amp, bias, completionFunction)
  i_env = Atransient(env2, duration, index, 0, 'dspRemove');

  -- args (table, dur, amp, bias, completionFunction)
  a_env = Atransient(env1, duration, 0.8, 0, 'dspRemove');

  {
    (osc.value(\, \, i_env.value) *! a_env.value).out(L).out(R);
  }.dspAdd(1);
}
```

Figure 5. SC Text View

It is important to remember that the text of a program is not the only contents of a SC file; the file also includes the definitions of wave tables, envelopes, etc.

Message Transcript

The message view (or transcript) shows messages from the compiler, and user programs can write to it with error or status strings from within an instrument using the `post` message (sent to a string, symbol, list, or number). Error messages are written here by the compiler (see the figure below for an example).

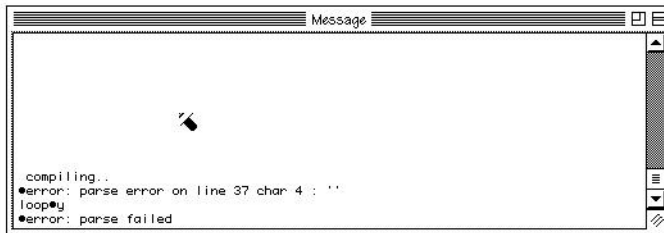
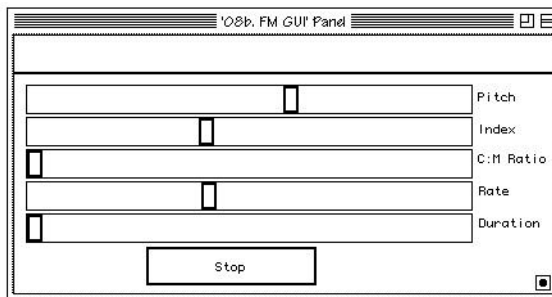


Figure 6. Message View (Transcript)

Instrument GUI

The third window that is always open when using SC is the current program's graphical user interface (GUI). The view may have a variety of sliders and buttons that control the program's instruments at performance time. There is a straightforward editor for creating and modifying SC GUIs, and you can read or set the values of GUI elements from your own programs easily. By default, this view is empty.



The example shown to the left is a “control panel” for a simple FM instrument; with this, the user can vary the pitch, modulation index, carrier:modulation frequency ratio, repetition rate, and note duration while the instrument is playing. You can create and edit a GUI for any instrument you write (see below).

Figure 7. Example of an Instrument GUI

When the run-time system is executing (i.e., when you're playing sounds), the top part of the instrument GUI (its header, shown as a blank area in the figure above) displays the percentage of the CPU cycles that are used for computation and for garbage collection, respectively in the top row, and the time clock, output level, and clipping indicators in the bottom row. The format of this display is shown in the figure below.

CPU: 22.78 %	■ GC: 3.91 %	
00:00:35.61	Vol: 0.00 dB	Out: L R clip

Figure 8. Example of an Instrument GUI Header

Table Views

The user can view and edit a program's envelope functions, wave tables, and audio buffers using the various table views. One can create tables using break-points, Fourier summation of sinusoidal overtones, or several other methods (see the **Fill** menu). Three different kinds of tables are shown in the figures below: a wave table created using Fourier summation, an envelope drawn by hand, and a break-point envelope. For each of these kinds of functions, you can edit the values, either using the automatic generation techniques just listed, or by hand-drawing break points in the case of break-point envelopes.

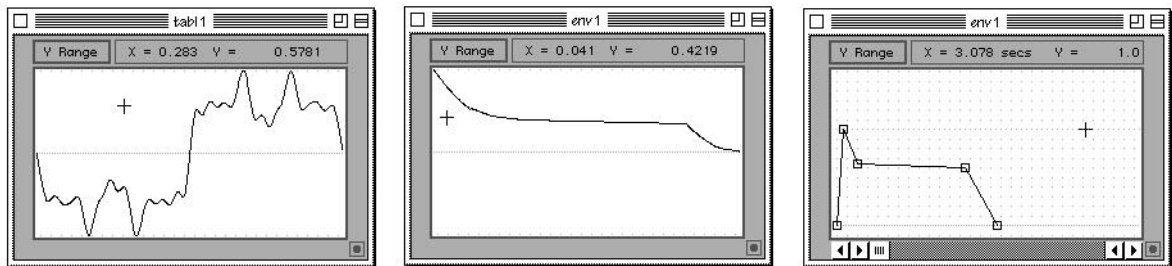


Figure 9. Example Table Views

There are several special dialogs for creating and editing wave tables, envelopes, and break-point functions. The figure to the left shows the interface for using Fourier summation, with which one can select the amplitude and phase of a number of sinusoidal partials.

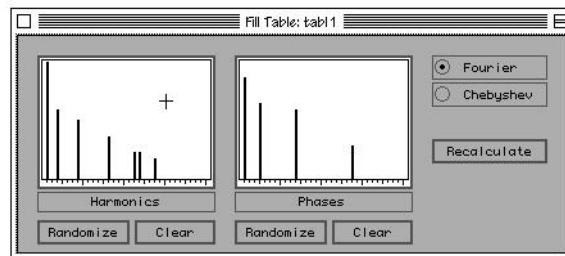


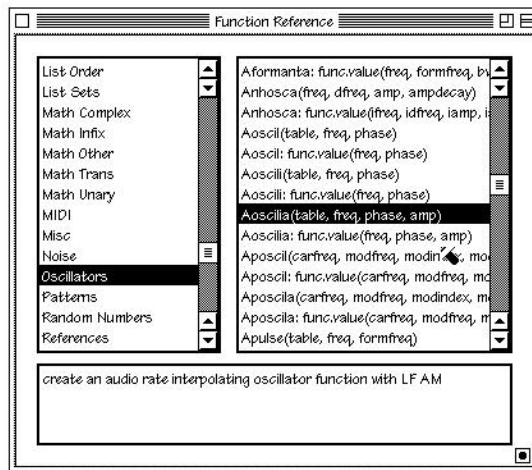
Figure 10. Table Editor for Fourier Summation of Sines

Other techniques include using FM to generate wave tables, making smooth envelopes for granular synthesis, various noise generators and a band-limited pulse generator. The break-point envelope editor shown in the right-most view of Figure 9 allows you to drag points with the mouse, or to add new points using **command-click** with the mouse.

Function Reference Help View

Typing **Command-t** brings up a very useful command reference window. It is shown in the figure to the left and is a very powerful tool for the novice and seasoned SC pro-

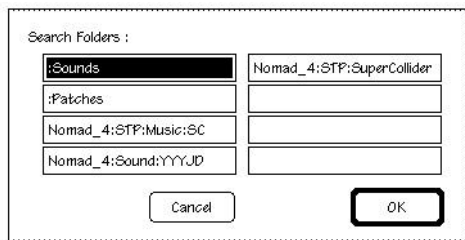
grammer alike. The list on the left of this view shows the “categories” of built-in functions, and the list on the right shows the functions in the chosen category (oscillators in the Figure). Note that the right-hand list also gives the function arguments for the constructor function and value message for each type of unit generator.



The first category in the left-hand list (not shown in this figure) is “all” for the case that you do not know where a particular function is categorized. Selecting a function name in the right-hand list displays a short comment about it in the lower text view (as in the figure). Unit generators provide two entries (as shown for the oscillator examples): one for the the arguments of the constructor method, and a second giving the arguments of the value message (which may be different from the constructor).

Figure 11. Function Reference Help View

Setting Search Paths



There is a dialog box that allows the user to set the list of directories where SC searches for sound files and instrument patches. The menu item **Set Search Paths** can be found in the **Synth** menu; it brings up the dialog view shown on the right. The default paths are the `:Sounds` and `:Patches` folders that are part of the SC distribution. Users can add their own sound file folders to this list by typing in to the other text fields.

Figure 12. Set Search Paths Dialog

Setting Global Values

There are a number of global values that can be set for the SC execution environment. The sub-frame size is analogous to the “control rate” in other MusicN languages. It is the size of the sample block that is used for the calculation of envelopes. Making it smaller may improve the sound quality, but will also consume more compute resources. The frame size is the length of the final output buffers that are sent to the sound output manager. Making this smaller will improve interactivity, at some computational cost. The dialog box shown on the right can be opened with the menu item **Set Globals** in the **Synth** menu (or with the command-key combination **Command-g**).

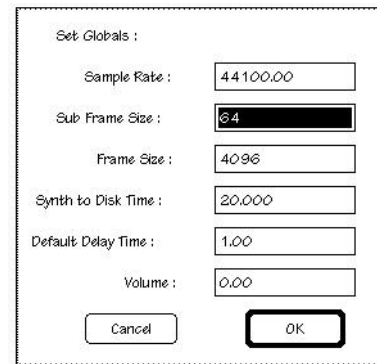


Figure 13. Global Settings Dialog

The sample rate is usually 44100 Hz, but 22050 or 11025 are also useful values. The control (sub-frame) size is 64 by default, and can be set to values between 4 and 512 (typically a power of 2 is used). The output buffer frame size is initially 4096, but can be set to values between 1024 and 16384 (also a power of two).

4.2. SuperCollider Menus

As is typical of Macintosh applications, the most common SC interaction functions are accesses with pull down menus found along the top of the screen. The standard Macintosh command-key accelerators are implemented here, and are indicated in the menus.

The *File* menu includes items for opening and saving SC program and data files, as well as for quitting SC.

The *Edit* menu contains the standard text editing functions, as well as special operations for inserting several kinds of delimiters (such as comment delimiters) around the selected text.

The *Synth* menu's items allow you to compile and play your program, to set the global values introduced above, and to set the output and input connections (i.e., to read sound input from the sound manager or from a file).

The *GUI* menu enables one to edit the GUI is an SC program, and, when in edit mode, to add user interface items to a screen.

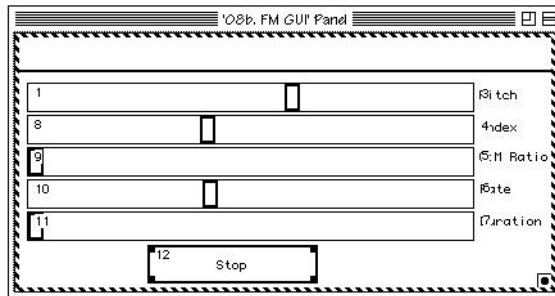
The *User* menu is empty by default, but there are SC functions with which advanced users can add their own items to this menu and bind them to SC functions. (This is outside the scope of this tutorial.)

The *Fill* and *Alter* menus are used with the table and envelope editors to create or modify wave tables or function envelopes.

The last three menus—*Tables*, *Envelopes*, and *Audio*—provide access to an SC program’s defined wave tables, break-point envelopes, and audio buffers, respectively. If your program declares any of these, its name will appear in the corresponding menu, and selecting it there will open a table editor on it.

4.3. Editing Instrument GUIs

SC includes a simple editor for laying out graphical user interfaces (GUIs) to instruments, or in fact, arbitrary programs. To enter the edit mode, type **Command-e**.



GUIs can have sliders, buttons, text fields, and other kinds of elements in them, and these can be read from, or written to, from within SC programs. Each element of a GUI has a numerical index (shown at the left edge of each element in the figure); these indices are used with the messages `getItemValue` and `setItemValue` to read/write the GUI item.

Figure 14. Instrument GUI in Edit Mode

When you enter edit mode, the border of the GUI window will change as shown in the figure above. When in edit mode, you can add new GUI elements, move or resize existing ones, or edit items’ properties. The resize button in the lower-right of the GUI window can be used to resize the window.

Standard GUI Items

SuperCollider GUIs can involve a variety of items that support the manipulation of numbers, numerical ranges, texts, or action triggers. There is no “drag and drop” or score-based GUI at present.

The standard GUI items are listed in the table below along with their common usage.

Type	Usage
------	-------

Button	Sends a message whenever pressed (that can be handled asynchronously by a function in your program)
Radio Button	Allows the user to select one item from several options
Check box	Toggles a value (that you can check from within the program) on and off
Slider	Sets a numerical value in a given range (with a given step size)
Range slider	Sets upper and lower limits for a numerical range
Label	Display a string (e.g., a label for another GUI item)

Table 13: Types of SuperCollider GUI Items and Their Uses

GUI Editing

To add a new GUI element to an SC GUI window, type **Command-**, (comma); the cursor will change to a cross-hair and you can draw a rectangle in the GUI window to set the size and position of the new item.

Once you have done this, you will be presented with a dialog box like the one shown in the figure on the right in order to edit the new item's type and properties. When you select a specific type of GUI item with the pull-down menu labeled "Type:" then the fields in the rest of the dialog box will change to those that are relevant for the chosen type of GUI element. In the case shown in the figure, we can edit the properties of a numerical slider: its minimum and maximum value, default value, and step size.

GUI Item Number: 13

Type: Slider

Resize: Fixed Left Fixed Top

Mouse: Update:

Min: 0

Max: 1

Value: 0

Step: 0

☐ continuous mode

Cancel OK

Figure 15. GUI Element Dialog Box

If you choose to add a range slider, there are two values that you can edit—the upper and lower bounds of the numerical range. For buttons, the important aspects will be label of the button and the message that is sent when the user presses it (this will be the name of a function that you have implemented in your program). For a stand-alone label string, you can type in a string.

To change the properties of a GUI element, select it (whereupon it will display resize handles in its corners) and type **Command-i**. To duplicate an item, select it and type

Command-d. To exit GUI edit mode, type **Command-e** again. The connection between the GUI elements and your SC program will be determined by how you implement the messages that are sent from buttons you might place on the GUI, and what you do with any numerical values the user can set, as we'll see in the examples that follow in the next Part.

4.4. Special Command Keys

Several special Macintosh Command-key combinations are used in SC. The most useful of these are listed below.

Command-/	Compile and play the current program.
Command-k	Compile (but don't run) the current program.
Command-.	Stop playing (interrupt).
Command-[Decrease volume 3 dB.
Command-]	Increase volume 3 dB.
Command-t	Bring up function reference help view.
Command-g	Bring up global settings dialog box.

Table 14: Special Function Keys

Part 5. Essential SuperCollider Programming

In this part of the book, I will present a series of progressive examples of SC sound synthesis instruments and signal processing programs. We start with simple instruments and then move on to discuss envelopes and control, extended synthesis techniques, the development process, and other topics. Readers who are unfamiliar with the synthesis techniques discussed here are referred to (Roads 1996).



5.1. Building Synthesis Programs

Some of the examples that follow are taken from James McCartney’s manual or on-line examples (with comments added). The text and the comments in the code are intended to complement each other, and you are encouraged to read them both.

A Wave Table Oscillator

The first example illustrates a simple “steady-state” chorusing oscillator (which is actually two oscillators in parallel that are slightly detuned, so that they beat in and out of phase, leading to a chorus-like effect). The wave form is defined using the wave table editor; it is not stored in the text part of the program. As in the example given at the start of this document, there is no envelope—this is a steady-state instrument that will play a 440 Hz tone with a 2 Hz beat frequency to both output channels.

```
-- Chorusing wave table oscillator (See the SC manual p. 55.)
-- This simple example plays a continuous tone with a “chorusing” (beat frequency) effect.

defaudioout L, R;          -- Declare audio outputs named L and R.
deftable table1;           -- Declare a stored wave table for oscillator.

start {                    -- This function gets called automatically; it does all the work.
  var osc;                 -- Declare the variable name “osc” for the oscillator.

                           -- Create a chorusing wave table oscillator object.
                           -- Acoscili’s arguments are (wave table, freq, beats/sec).
  osc = Acoscili(table1, 440, 2);

  {                        -- Create a closure for the DSP loop; it gets osc’s value,
                           -- scales it (*! is an in-place multiplier), and sends it out.
    (osc.value *! 0.2).out(L).out(R);
  }.dspAdd;               -- Add the closure to the DSP loop
}                          -- End of start(); end of the program
```

Code Example 14. Continuous Chorusing Oscillator Instrument

In the code example above, the first two lines are comments. The first executable statement declares sound output buffers named *L* and *R*; these will typically be connected to the Macintosh sound manager to play sound out to your loudspeakers. (You can also play out to a sound file.) The deftable declaration defines a wave table named *table1*. Declaring it does not give it any contents though. The start function is the body of the instrument; if there is a function named *start* in your SC program, it will be called when you start execution. Thus, you can write any statements you want automatically executed in the start function.

In this case, the entire instrument is placed in the start function. It consists of (1) the declaration of the variable named *osc* (which, as you remember, is untyped, it could be an oscillator, or any other kind of object), (2) an assignment that creates an oscillator (using a constructor message), and (3) the DSP loop. The name of the oscillator constructor—*Acoscili*—denotes an audio-rate (A), chorusing (c) oscillator (*oscil*) with sample interpolation (i). The DSP loop is simply a closure (the statement enclosed in curly braces) to which we send the message *dspAdd*. This will be executed continuously when we play the instrument. In the loop, we send the *value* message to the oscillator object, scale its output by 0.2, and send that data buffer to both outputs in succession (with the cascaded out messages). The last curly brace ends the start function, and thus the whole program (for this simple case).

If you were to develop this instrument yourself, you would type in the text shown above, then compile it (either from the corresponding menu item in the **Synth** menu or with **Command-k**). After you compile it, the wave table you declared (with variable name *table1*) would appear in the **Tables** menu, from which you can open a table editor and change its contents. (To do this, open the table editor, then use the **Fill** menu to edit it; for the start, try using Fourier additive synthesis and creating a wave table with a few different overtones.) Once you have a wave form defined, you use the key combination **Command-/** to play the instrument (i.e., execute the start function and run the DSP loop); typing **Command-.** will stop the playing.

Note also the use of the *!** operator. This is a multiplier (like ***), but does its work in-place, i.e., it overwrites its receiver with the result. This means that the value of the oscillator (a buffer of 64 samples) will be scaled by 0.2 in-place, which will be faster and more space-efficient, but might be dangerous (in the case where we want to do something else with the un-scaled sample buffer later). If you want to visualize this instrument, the relevant graphical elements would be the oscillator, its parameter inputs, the amplitude scaling, and the outputs.

In SC, not all oscillators have separate amplitude inputs (in this example, it generates full-scale output), so I draw it with an amplitude input of 1. The resulting flow chart is shown in the figure on the right. This example demonstrates most of the basics of SC programming: the importance verbose and well-formatted of comments, the necessary declarations for the output buffers and tables, and a rudimentary start function that includes a DSP loop. We will extend this through several stages in the progressive examples that follow.

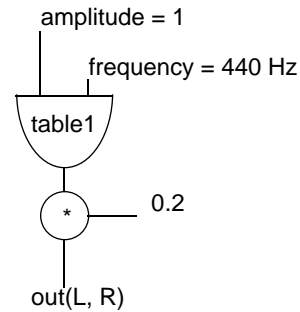


Figure 16. Visualization of the Chorusing Oscillator

Using an Envelope

The second example demonstrates a repeating rhythm. I'll define an instrument that plays the wave table oscillator of the above example (using a random frequency of $[100 + 200.\text{rand}]$ —something between 100 and 300 Hz) and an envelope (defined by the function `env1`).

To make the envelope, we have to define its shape (i.e., put something into the table `env1`), use the right unit generator constructor to get an envelope object, and multiply the output of the oscillator by the output of the envelope generator inside the DSP loop.

The envelope unit generator is an instance of `Ktransient`—a control-rate transient generator. The arguments of its constructor are the table to read through (once per note), the duration of the note, the amplitude (usually 0-1), a bias value (or offset, usually 0), and a *completion function*. The completion function is a reference to the function that the envelope generator should call when it terminates. Using ``dspRemove` as the completion function means that the instrument will be removed from the DSP loop at the end of the envelope. This is standard practice; most envelopes take ``dspRemove` as their completion function (though you could call any other function).

The DSP loop of this instrument sends the value message to the oscillator (to get a buffer of samples), and then multiplies that buffer by the result of sending the value message to the envelope generator (which will return a single value because it is a control-rate generator). This means we'll have a new envelope value every 64 audio samples. The result of this multiplication (the scaled sample buffer) is then sent to both outputs.

The last line of the program is a list containing the number 0.5 and the special variable `thisFunc` (a reference to the current function—`start`). We send the message `sched` to the

list, which means that the system should reschedule the start function (thisFunc) after 0.5 seconds, i.e., the note will repeat twice a second. Because the duration of the notes is set to 1 second (the second argument in the Ktransient constructor), and we repeat then every 0.5 seconds, this will be *molto legato* (or 200% duty cycle).

-- Wave table oscillator with repeat. (Taken from the SC manual p. 56.)

```
defaudioout L, R;          -- Declare outputs.
deftable tab1, env1;        -- Declare 2 wave tables--one for the envelope.

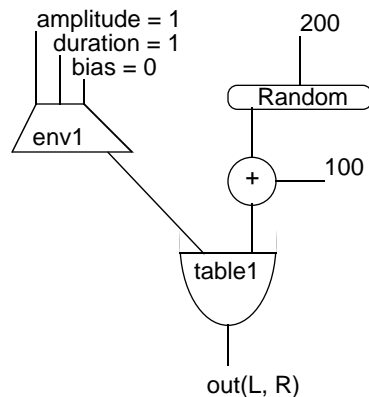
start {                    -- Start function is the repeating instrument.
    -- Signal oscillator.
    -- Arguments are (table, frequency, beats/sec).
    osc = Acoscili(tab1, 100 + 200.rand, 2);

    -- Envelope transient function (Ktransient).
    -- Its arguments are (table, duration, amplitude, bias, and a
    -- completion function [function reference]).
    amp = Ktransient(env1, 1, 0.2, 0, `dspRemove);

    -- DSP loop closure.
    -- Scale the oscillator by the envelope.
    { (osc.value *! amp.value).out(L).out(R) }.dspAdd;

    -- Repeat the note in 0.5 seconds.
    [0.5, thisFunc].sched;
}
```

Code Example 15. Chorusing Oscillator with an Envelope



The envelope function `env1` is created and edited like an oscillator wave table. Look at it in the table editor and you'll notice that it does not look like a wave form, but more like a note envelope (remember Figure 3 above). The visual flow chart for this instrument would now have to include the envelope generator (drawn with a different icon than the oscillator, and showing its inputs) and the random number generator for the frequency (including its scale). This might be drawn as shown in the figure to the left, which uses different icons for the different unit generators.

Figure 17. Flow Chart of an Oscillator with an Envelope

The new features of this instrument are the envelope (and the use of two unit generators with their respective value messages in the DSP loop), and the periodic repeat achieved using a two-item list (with a scheduling time and function to call) and the `sched` message.

As an exercise, you can type **Command-g** to open the global variables dialog and change the sub-frame size (control rate) to a smaller value such as 8. You should notice that the sound quality improves and the computation percentage increases.

Playing a Score

To show you how to make a more controllable musical experience, let us introduce a note list (score) to the example. This process will consist of rewriting the instrument above as a separate function that takes its duration, pitch, and amplitude as function arguments. In the following code, we have simply substituted function arguments for the fixed constants of the previous version.

```
chorus_instr {
    arg dur, pitch, amp;
    -- Make a new function named chorus_instr.
    -- Declare the arguments of the instrument.

    -- Signal oscillator; args are (table, freq, beats/sec)
    osc = Acoscili(tabl1, pitch.midicps, 2);

    -- Envelope transient function
    -- args are (table, dur, amp, bias, completionFunction)
    amp = Ktransient(env1, dur, amp, 0, `dspRemove);

    -- DSP loop -- send (osc * env) to both outputs.
    { (osc.value *! amp.value).out(L).out(R) }.dspAdd;
}
```

Code Example 16. Chorusing Oscillator Written as a Function

Note the use of the `midicps` message in the oscillator constructor above. This converts between MIDI key numbers (pitches) and frequencies in Hz (cycles-per-second or cps). This will allow us to use MIDI key numbers in our score, which will be converted to Hz inside the instrument.

With this function defined, we can make a start function that calls it and supplies parameters in the manner of a MusicN-style note list. To do this, we use the `sched` message (sent to a list) that was introduced above, with one important extension. To play a simple scale, we would write the following.

```

defaudioout L, R;          -- Declare outputs.
deftable tabl1, env1;      -- Declare 2 wave tables--one for the envelope.

start {                    -- Play a score in the start function

--   time   instrument   dur pitch amp
[0.00, 'chorus_instr, 0.25, 48, 0.5].sched;
[0.25, 'chorus_instr, 0.25, 50, 0.5].sched;
[0.50, 'chorus_instr, 0.25, 52, 0.5].sched;
[0.75, 'chorus_instr, 0.25, 53, 0.5].sched;
[1.00, 'chorus_instr, 0.25, 55, 0.5].sched;
}

```

Code Example 17. Score for the Chorusing Oscillator Instrument

Here we use the list with [start-time, function-reference] as in the previous example, but we add extra list elements after the function reference. These are passed as arguments to the function (instrument) `chorus_instr` when it is scheduled, so they become the durations, pitches, and amplitudes of our notes, respectively (since these are the three arguments of the instrument definition function).

We have not presented the basics of instrument construction, plugging together of unit generators, and score writing. The next chapter will step through another example showing the recommended process of instrument development, debugging, and GUI construction.

5.2. Developing an Instrument

FM With Repeat

To illustrate the typical process of instrument development, and another synthesis method, we will step through three versions of a frequency modulation (FM) instrument, going from the simplest working example, to one with a GUI to control its parameters, to a version that uses a score for performance.

This instrument uses an FM oscillator (actually it uses phase modulation—a subtly different version of the FM equation) with two different envelopes for the modulation index and amplitude function. In this first version, we will make a repeating version where all parameters are set to be constants with small random variations.

```

-- FM instrument with repeat
-- We use two envelope functions for the amplitude and modulation indices.

defaudioout L, R;          -- Declare the outputs.
deftable tabl1, env1, env2; -- Declare the wave tables and envelopes.

```

```

start {
    -- The start function is the instrument.
    -- Declare the 6 local variables.
    var freq, index, ratio, osc, i_env, a_env;
    -- Set up the parameters for debugging.
    -- (These are Constants + random ranges.)
    freq = 200 + 200.rand;
    index = 1 + 2.rand;
    ratio = 1 + 0.2.rand;
    -- Frequency between 200 and 400.
    -- Mod. index of 1, 2, or 3.
    -- C:M frequenct ratio of 1 to 1.02 (inharmonic)

    -- FM oscillator constructor
    -- Its arguments are (carrier_freq, mod_freq, index).
    -- (Index = 0 for now.)
    osc = Aposcil(freq, freq*ratio, 0);

    -- Amplitude envelope (env1 is the envelope function).
    -- Arguments are (table, dur, amp, bias, completionFunction).
    a_env = Ktransient(env1, 1, 0.8, 0, `dspRemove);

    -- Modulation index envelope (env2 is the envelope function).
    i_env = Ktransient(env2, 1, 1, 0, `dspRemove);

    {
        -- DSP loop, plug in index envelope (as 3rd arg in value()),
        -- and multiply by amplitude envelope.
        (osc.value(\, \, i_env.value) *! a_env.value).out(L).out(R);
    }.dspAdd;

    -- Repeat a note at 2 Hz
    [0.5, thisFunc].sched;
}

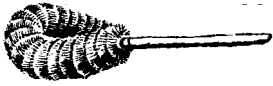
```

Code Example 18. FM Instrument—First Pass

In this example, we defined the two envelope tables (if you're on-line, look at them in the table editor), and used them both in different Ktransient unit generators (two instances of the same class). The DSP loop sent the value message to the oscillator and plugged in a new value for the third argument (the modulation index envelope). This instrument will note play two overlapping notes per second, each with a different pitch, modulation index, and carrier-to-modulation frequency ratio.

This version of the code has allowed us to get the simplest version of the FM instrument working, though we do not have very much control over it in real time. If you had problems getting it to work, you would be better off trying to debug a simple version than a more complex one.

Debugging SuperCollider Programs



There are a number of techniques that will make it easier for you to get your SC instruments to work as you planned them. The first two techniques constitute what I like to call *antibugging*., and the rest are derived from common sense

Start Small

The structure of this example is intended to demonstrate the incremental development approach. It is much easier to debug a program that's 20 lines long, and then add ten new lines per debugging stage than to wait until it's too long to fit on one screen. You can also compile your program at any time (to check for syntax errors) without actually running it using the **Command-k** keys. The message transcript view will show any syntax errors you might have (and the compiler will place the text insertion cursor right where it thinks the error takes place).

Use post and dump Where Appropriate

Some C programmers call this “debugging by printf.” The technique simply means that you can trace the flow of your program by using the post message to write strings or numerical values to the message transcript view from within your code. This often helps to find out where your program fails (i.e., posting messages about what functions are being called when), or why it doesn't sound the way you expect it to (i.e., posting messages to the transcript about what the parameter values are).

The post message is normally used with simple variables such as strings, symbols, numbers, and 1-dimensional lists; dump can be sent to complex objects such as multi-dimensional lists and unit generators to display their inner state variables.

Document Your Code Profusely

I generally use the rule that there should be more English (or whatever your native language happens to be) than SC in well-written programs. Although the examples in this book are quite verbosely commented, I recommend that there be a one-line comment for every line of program code that is not extremely trivial. This is not only valuable for you, but it can help a friend to debug your code (i.e., find out where the code doesn't do what the comment says it does).

Save Your Work Frequently

SuperCollider is an excellent and stable programming environment; nevertheless, it does crash from time to time. Murphy's law dictates that the probability of a crash is proportional to the square of the time between file saves. Use **Command-s** frequently to save your program to the disk.

Write show Functions

We will present an example below of an instrument whose GUI has a button that calls a function to display the instrument's parameters in the transcript view. This kind of "brain dump" function that is sent from a GUI button can be very useful as a debugging tool (i.e., "what are the parameter ranges where the instrument distorts").

Use the init Function to Test Expressions

As described above, the function `init` (if you define it) gets called whenever you compile or start an SC program. You can treat this as a simple "SC interpreter" and use it to test out SC expressions. If you type an expression into the `init` function and then type **Command-k**, the expression will be executed. I use this frequently together with the post message to see the result of simple SC statements (e.g., `init{ (21.334 * 72.1149).post }`).

Practice Modular Design and Reuse

The most important and powerful debugging technique is to steal working code (rather than writing your own buggy code) wherever possible. This is also a large part of the motivation for the development of this book, and one of the wonderful benefits of the large program library that is shipped with every copy of SC. (Thanks, James!) I recommend that you also get one of the several Macintosh programs that can search the contents of text files, for example the ShareWare SearchFiles program by Robert Morris. This makes it easy to search among the demonstration code for examples of the usage of a particular function or keyword (e.g., "find me examples of case statements in instruments").

Read the Message View

The SC compiler prints terse but meaningful error messages in the transcript at compile time and at run-time. Read them.

SuperCollider Error Messages

In general, there are several kinds of compile-time and run-time errors, and these are generally caught by the SC system and reported in the message transcript. I'll introduce the most common of these below and give basic examples.

Reference Errors

Many compiler errors are based on simple typing mistakes and are identified and reported very well by the SC compiler. A reference to a non-existent function or variable, for example, earns you the message,

- error: Function or Method "Assinosc" is not defined.

Expression Syntax Errors

There is an endless variety of SC expression syntax errors (that I'm in the midst of exploring). Many of these are caught and reported by the SC compiler. In most cases, the compiler prints a message in the transcript view telling you the exact location where the parser failed in reading your code, as in the following example.

```
x + 4 = 2;           -- Very illegal syntax: a complex expression on
                     -- the left-hand side of an assignment
```

In this case, the error message in the transcript reader

```
•error: parse error on line 10 char 7 : '='
x + 4 = 2;
```

The line/character numbers and the “•” in the second line of the error message identify the exact position where the parsing failed, giving you a strong hint to look just to the left of it for questionable syntax. In some cases, the parsing fails on line *x* because you forgot to terminate line *x* - 1 with a semi-colon (this is one of my personal favorites).

Uninitialized Variables

Because SC does not require variable declarations, the compiler does not always catch simple typing errors when they first occur. Examine the following code fragment.

```
var osc;              -- Declare the variable osc.
oss = Asinosc(200);   -- Assign an oscillator to “oss” (in error).
                     -- The compiler will auto-declare oss for you,
                     -- and leave osc as nil.
{ osc.value.out(L).out(R) }.dspAdd;  -- This works, but produces no sound!
```

In this case, we get no error messages, but also no sound. This is a good argument for the anti-bugging techniques described above. You could catch this if you had executed `osc.dump(2)` just before the DSP loop to show you the internal state of the uninitialized variable that you think should be an oscillator object.

Message not Understood

If you send a message to an object that does not understand it, the error message is quite clear; for example the expression,

```
x = “hello”.min;      -- Take the minimum of a string (?!?!)
```

generates the error message,

- error: type: string does not understand: 'min'

Numerical Range/Domain Errors

SC tries to be flexible in handling such numerical errors as division by zero (answers negative infinity), integer overflow (answers +-1 (?)) or taking the square root of a negative number (answers a complex number).

Running out of Tasks

The task scheduler can hold at most 512 tasks, so if you use the sched message to put task in the queue, you might see the message,

- error: Too many active tasks, Priority heap full!

If this is the case, you'll have to write a smarter scheduling function that waits a bit and then schedules more events incrementally.

Building a GUI for a Stereo FM Instrument

We will now return to where we left off on the development of our FM instrument. In the next step (once we have the basic instrument debugged), we make a version where all the important parameters can be set interactively from sliders on a GUI. Using the GUI editor, the sliders were given labels and meaningful numerical ranges. The GUI screen layout is shown in the figure below. (On-line readers should edit the GUI to see the settings of the items; enter edit mode, pick a slider, and use **Command-i** to get its information dialog.) Knowing what the number of each GUI slider is, we can use the `getItemValue` message within the instrument to get its value.

The other important addition to this version is the use of a low-frequency oscillator (LFO) to control the stereo position. We use a polled sine oscillator (Psinosc) that we evaluate in the `pan2out` stereo output panner function; it returns a single value each time we call it. We use this to pan the FM instrument slowly between the output channels as shown in the example below.

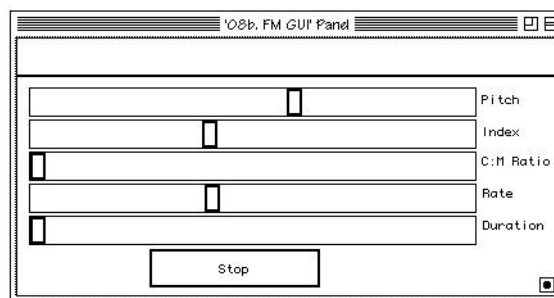


Figure 18. FM Instrument GUI

```

-- FM oscillator with repeat, GUI input, and LFO panning.
-- All parameter values set from the GUI sliders.
-- Slider numbers are:
--      1 (freq),  8 (index),  9 (c:m ratio),
--      10 (repeat rate), and 11 (duration).

defaudioout L, R;          -- Declare outputs.
deftable tabl1, env1, env2; -- Declare wave tables and envelopes.

start {                    -- Declare local variables.
    var freq, index, ratio, rate, duration, osc, i_env, a_env, lfo;

                                -- Get and scale the GUI slider values.
    freq = 1.getItemValue + 20.rand; -- Frequency is slider 1; add some random variation.
    index = 8.getItemValue;          -- Modulation index is slider 8.
    ratio = 9.getItemValue;          -- C:M frequency ratio is slider 9.
    rate = 1 / 10.getItemValue;      -- Repetition rate is 1 / slider value.
    duration = 11.getItemValue;      -- Note duration is slider 11.
    -- ['Freq: ' freq ' Index: ' index].post; -- For debugging, uncomment this.

                                -- Unit generator instance creation messages.
    osc = Aposcil(freq, freq*ratio, 0); -- FM oscillator.
                                -- Two envelope generators (amplitude and index)
    a_env = Ktransient(env1, duration, 0.8, 0, `dspRemove);
    i_env = Ktransient(env2, duration, index, 0, `dspRemove);

    lfo = Psinosc(0.2, 0);        -- Define a polled sine LFO at 1/5 Hz for panning.

                                -- DSP loop. (the pan2out() message takes a position
                                -- value (+-1) and two output channels).
    { (osc.value(\, \, i_env.value) *! a_env.value).pan2out(lfo.value, L, R) }.dspAdd;

    [rate, thisFunc].sched;      -- Repeat notes at the chosen rate
}

```

Code Example 19. FM Instrument that uses GUI Sliders for Parameters

With this GUI-driven version of the instrument, we can experiment with its parameters, finding the useful ranges of the various parameter values and the palette of different sounds it can make.

FM with a Score

As the final stage in our development, here's a version of the FM instrument that is written as a separate function and then called from a score note list declared in the start function. One can call more than 1 instrument in this way, or read score data from a file (I'll show you how later). The only limitation is that one can schedule no more than 512 events in the future, so you need to write a more sophisticated score reader for long

scores. We have also added a pitch-to-frequency mapping here in that the instrument's argument `freq` is sent the message `midicps`.

-- FM instrument with a score (to play an arpeggiated C-minor chord).
 -- The instrument definition is written as a separate function, and the start function calls it.

```
defaudioout L, R;
deftable tabl1, env1, env2;

start {
    -- Start() calls the instrument(s).
    -- time  instr.  dur freq ind ratio
    [0, `fm_instr, 1, 36, 2, 1.02].sched;
    [0.5, `fm_instr, 1, 39, 2, 1.04].sched;
    [1, `fm_instr, 1, 43, 2, 1.02].sched;
    [1.5, `fm_instr, 1, 48, 2, 1.04].sched;
    [2.6, 'dspKill].sched; -- Kill the DSP loop at the end of the score.
}

fm_instr{
    -- Simple FM instrument written as a function.
    -- Arguments: (duration, pitch(!), mod. index, and c:m ratio)
    arg duration, pitch, index, ratio;
    -- Declare local variables.
    var osc, i_env, a_env, hz, lfo;
    -- Convert the pitch to Hz.
    hz = pitch.midicps;
    -- Unit generator instance creation (same as above).
    osc = Aposcil(hz, hz*ratio, 0);
    a_env = Ktransient(env1, duration, 0.8, 0, `dspRemove);
    i_env = Ktransient(env2, duration, index, 0, `dspRemove);
    lfo = Psinosc(0.2, 0);
    -- DSP Loop.
    { (osc.value(\, \, i_env.value) *! a_env.value).pan2out(lfo.value, L, R) }.dspAdd;
}
```

Code Example 20. FM Instrument as a Function with a Score

In this Part, we stepped through the development process using the example of a simple FM instrument. The following chapters will introduce more interesting synthesis techniques and more complex instruments.

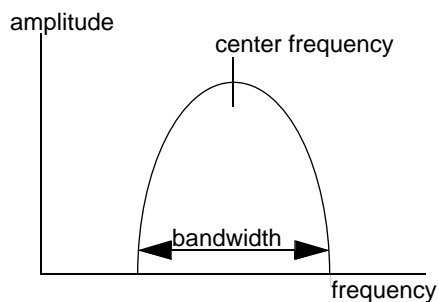
5.3. Other Sound Synthesis Techniques

The next few sections present several more interesting sound synthesis techniques and SC programming paradigms. Readers unfamiliar with the synthesis methods are referred to (Roads 1996) for introductory material.

A Filtered Noise Instrument

SuperCollider has a full complement of noise generators and dynamic digital filters, with which we can construct a filtered noise instrument as shown below. In this example, we introduce band-limited noise, a filter unit generator that takes its signal input from another unit generator's output, the use of break-point envelope functions (which we will discuss in more detail in a later chapter), and panning a signal between the stereo outputs. Filter-based instruments, which are usually groups together into the class of subtractive synthesis because they start with a spectrally rich signal and use a filter to subtract components from it.

In the example we present first, we take a broadband noise signal and use a band-pass filter on it. This filter will filter out both the very high-frequency and the very low-frequency components of the noise signal, leaving only those components that are near to its center frequency. The figure below shows an idealized frequency-vs-amplitude graph (frequency response) of a band-pass filter; on it, you can see the center frequency and the bandwidth of the filter.



In the example code that follows, you'll see the use of a noise unit generator, the "patching" of the noise input to the filter within the DSP loop, and the way we create break-point envelopes. The envelope function table for a break-point envelope is declared using `defenvelope` rather than the `deftable` that we used above together with `Ktransient`. The filtered noise instrument is written as a stand-alone function that is called from two note commands in the program's start function.

Figure 19. Frequency Response of a Band-Pass Filter

```
-- Stereo low-pass filtered noise instrument.
-- This uses a separate instrument function and plays two notes from the start function.

defaudioout L, R;          -- Define outputs.
defenvelope env_func;      -- Define a break-point envelope function.

start {                     -- Play 2 notes from within start().
--   start instr    dur amp freq bw  pos
  [0, `noise_instr, 2,  1,  400, 20, -0.8].sched;
  [2, `noise_instr, 2,  1,  400, 8,  0.8].sched;
```

```

    [4.1, 'dspKill'].sched;    -- Terminate at the end of the score.
}

noise_instr {                  -- Band-pass filtered noise instrument written as a function.
                                -- Arguments are duration, amplitude, filter center frequency,
                                -- filter bandwidth, and stereo position (-1 to 1).
    arg dur, amp, freq, bw, pos;

    var noise, filter, env;    -- Declare the local variables

    noise = Anoise(sr/2.4, 1); -- Create a noise generator that goes from 0 Hz to SR / 2.4.
                                -- (Its amplitude is 1.)
    filter = Abpf(freq, bw);   -- Create a band-pass filter with the desired frequency and
                                -- bandwidth. Note that it has no input as of yet.

                                -- Create a control-rate break-point envelope.
    env = Kbpenv(env_func, amp, 0, dur, amp, `dspRemove); -- (Details will follow.)

                                -- Send noise through filter in DSP loop
    { (filter.value(noise.value) *! env.value).pan2out(pos, L, R) }.dspAdd;
}

```

Code Example 21. Band-Pass Filtered Noise Instrument

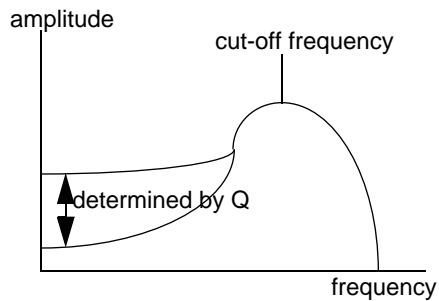
Three new unit generator constructor functions are used in this example. The arguments of the break-point envelope `Kbpenv` will be described in a later example. What's important to observe in the DSP loop is that the filter's value message takes the filter's input (i.e., the output of the noise generator) as its only argument. This is the first case we've seen yet where the arguments of value are different from those of a unit generator's constructor message. This is because the filter's input *must* be an audio-rate signal, and thus it cannot be set up in the instance creation call but rather has to be inside the DSP loop.

The `pan2out` message is sent to an audio-rate sample buffer (such as the output of the filter), just like the `out` function we've used up to now, but it takes a stereo position value as its first argument and two audio output channels as the other two. The position value can range from -1 to +1, and the function will mix the input signal to the two outputs according to that value (i.e., `pos = -1` means the whole signal goes to output L).

Filtered Noise with GUI Control

Newer version of SC (after 1.1b4) also include resonant filters, which are very useful for getting "that analog sound." These are demonstrated in the two following instruments, which also introduce some new GUI programming techniques.

In a *resonant filter* (compared to a typical low-pass filter), there is some amount of attenuation of the signal in the pass-band. Depending on the amount of this attenuation, the filter changes from low-pass to band-pass frequency response, as shown in the figure on the right.



The variable that determines this level is called the Q of the filter. Low values of Q (less than 2 or so) sound more like low-pass filters, but for higher Q values, the center frequency stands out more and the effect is like that of a narrow band-pass filter. As very high values of Q (greater than 50 or so) the filter begins to “whistle” and may even oscillate without any input at all.

Figure 20. Frequency Response of a Resonant Low-Pass Filter

The code for this example illustrates another case where a unit generator’s constructor message and its value message have different arguments; again, the filter is created with no input, and the input, as well as the center frequency and the Q , are “plugged in” inside the DSP loop.

```
-- Monophonic filtered noise instrument with GUI controls over the amplitude,
-- resonant low-pass filter frequency, and filter Q (sharpness).
```

```
defaudioout L, R;
```

```
start {
  noise_instr;          -- Play the instrument from the start function.
}
```

```
noise_instr {           -- Filtered noise instrument function.
  --Local variable declarations.
  var amp, freq, Q, noise, filter;
```

```
                        -- read the GUI values
  amp = 1.getItemValue; -- Item 1 is amplitude.
  freq = 2.getItemValue; -- Item 2 is center frequency.
  Q = 3.getItemValue;   -- Item 3 is filter Q.
```

```
                        -- Unit generator constructors.
```

```
  noise = Anoise(sr/2.4, 1); -- Broad-band noise generator.
```

```
  filter = Arlpf(freq, Q);  -- Low-pass filter with resonance.
```

```

-- Send noise through the filter and scale by amplitude
-- (read all 3 GUI sliders in the DSP loop--expensive).
-- in      freq      Q      amplitude
{ (filter.value(noise.value, 2.getItemValue, 3.getItemValue) *! 1.getItemValue)
  .out(L).out(R) }.dspAdd;
}

```

Code Example 22. Noise with a Resonant Filter and GUI

The GUI for this program has three vertical sliders that are labeled A (amplitude), F (frequency), and Q. In order to get real-time GUI control of a continuous instrument, we use the `getItemValue` message inside the DSP loop. This is quite computationally intensive and should only be used in cases such as this where you want immediate response from within a continuous instrument.

Noise and Buzz Resonances with GUI and Show Functions

The last example in this series uses two filter-based instruments in parallel. One of them is identical to the one given in the previous example, and the other uses an oscillator with a complex wave form and a resonant filter. Both instruments are played continuously and controlled from sliders on the GUI. In addition, both have stereo panning that is controlled from a GUI slider.

We have also added buttons to the GUI that send messages that cause the instruments' parameters to be posted to the message view, and another button that will stop the DSP loop. If you open this file on-line and edit the GUI, you can see that the buttons send the messages such as `showNoise`, which are implemented (but never sent) in the program.

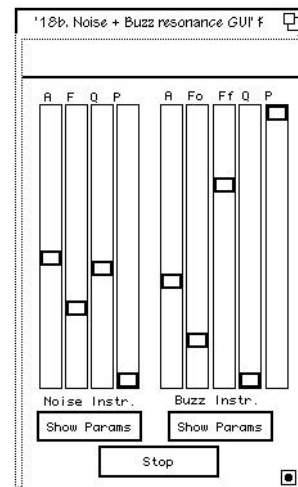


Figure 21. Noise and Buzz GUI

The last button—Stop—sends the message `dspKill`, which will halt the DSP engine as soon as it is sent. As `dspKill` is a built-in library function, we do not need to implement it in our program.

```

-- Filtered noise and pulse instruments with GUI controls and "show parameters" buttons.
--

```

```

defaudioout L, R;
deftable buzz;      -- Pulse-like wave table (edit it to see).

```

```

start {
    noiseInstr;    -- Play the instruments; they are controlled from the GUI.
    buzzInstr;     -- Play continuous filtered noise.
    buzzInstr;     -- Play buzz-like resonance instrument.
}

-- Resonant filtered noise instrument with GUI Control.
-- Sliders:
--     1 = amplitude, 2 = filter freq, 3 = filter Q, 5 = stereo position
--
noiseInstr {
    var amp, freq, Q, noise, filter;

    amp = 1.getItemValue;    -- Get slider values.
    freq = 2.getItemValue;
    Q = 3.getItemValue;

    noise = Anoise(20000, 1);    -- Noise generator.
    filter = Arlpf(freq, Q);    -- Band-pass filter.

    -- Send noise through filter in DSP loop.
    {
        --           input    frequency    Q           amplitude
        (filter.value(noise.value, 2.getItemValue, 3.getItemValue) *! 1.getItemValue)
        --           stereo position
        .pan2out(5.getItemValue, L, R) }.dspAdd;
    }

-- Resonant filtered pulse (buzz-like) instrument with GUI control.
-- Sliders:
--     6 = amplitude, 7 = osc. freq, 11 = filter freq, 8 = filter Q, 9 = stereo position
--
buzzInstr {
    var amp, freq, filt_f, Q, osc, filter;

    amp = 6.getItemValue;    -- Get GUI slider values.
    freq = 7.getItemValue;
    filt_f = 11.getItemValue;
    Q = 8.getItemValue;

    osc = Aoscili(buzz, freq);    -- Buzz oscillator.
    filter = Arlpf(filt_f, Q);    -- Band-pass filter.

    -- Send buzz through filter in DSP loop.
    {
        (filter.value(osc.value(7.getItemValue), 11.getItemValue, 8.getItemValue)
        *! 6.getItemValue).pan2out(9.getItemValue, L, R) }.dspAdd;
    }

-- Messages sent by "show" buttons.
-- Print slider values to the message view.
--

```

```

showNoise {
    'Noise Instrument'.post;
    ['  Filter frequency: ' 2.getItemValue].post;
    ['  Filter Q: ' 3.getItemValue].post;
    '.post
}

showBuzz {
    'Buzz Instrument'.post;
    ['  Oscillator frequency: ' 7.getItemValue].post;
    ['  Filter frequency: ' 11.getItemValue].post;
    ['  Filter Q: ' 8.getItemValue].post;
    '.post
}

```

Code Example 23. Resonant Filtered Noise and Pulse Instruments

The show functions defined above write lists to the transcript that include labels (the symbols that are the first items in the lists) and GUI item values. The results can be seen in the pretty screen dump on the cover of this book.

5.4. Making and Controlling Break-Point Envelopes

There are several kinds of transient time functions in SC. Up to now, we have used the simplest `Ktransient` function, which takes a hand-drawn data table and reads through it at a constant rate. There are, however, much more flexible means of making envelopes, which we will present now.



A break-point function is defined by a series of points (break-points) between which we interpolate, as in the traditional attack-decay-sustain-release (ADSR) envelope generator. In some MusicN languages, envelope functions all have the range and domain of 0 to 1, but in SC, they can have arbitrary ranges, and can be scaled within instrument programs. This means, for example, that you can draw a curve that represents a spatial trajectory, and give it real-time coordinates (i.e., set it to last 30 sec.).

To create a break-point envelope we use the `defenvelope` definition, and the `(AK)bpenv` unit generator. After you compile code that uses `defenvelope` (use **Command-k** to compile without executing) the name of the table you defined appears under the **Envelopes** menu for editing. In the envelope editor, one can change the “absolute” duration and scale of a time function, and add and move its break-points. “Grabbing” a point with the mouse allows you to move it, command-clicking adds a new point. Pressing the Y

Range button allows you to set the amplitude range of the function, and you can change its duration by dragging the right-most break point.

Basic Break-point Functions

The instrument below demonstrates the use of two different envelope functions within an instrument. The text of the instrument is simple, but the two envelopes *env1* and *env2* have quite different shapes (ASR vs. triangle-shape). If you plug one or the other of them in to the *Kbpenv* unit generator, you'll hear the difference.

```
-- Wave table osc. with two selectable break-point envelopes.
-- To see the envelopes, look at them via the "envelopes" menu.

defaudioout L R;
deftable tabl1;          -- Declare the oscillator wave table.

defenvelope env1, env2;  -- Declare two break-point envelopes.
    -- Envelope 1 has sharp attack, then lower sustain (as in a "traditional" ADSR)
    -- Envelope 2 is triangle. (Look at them both in the envelope editor.)
    -- They are both 2 sec. long (thus the time scale of 0.5 below)
start {
    var osc, env;
    -- Create a chorusing wave table oscillator.
    osc = Acoscili(tabl1, 100 + 100.rand, 1);
    -- Create a control-rate break-point envelope generator.
    -- table, amp, bias, timescale, completionFunction)
    env = Kbpenv(env1, 0.25, 0, 0.5, `dspRemove);
    -- Change env1 to env2 in the above statement to use the
    -- triangle-shaped envelope function.

    -- Get the oscillator's value and scale it by the envelope'.
    { (osc.value *! env.value).out(L).out(R) }.dspAdd;

    [1.0, thisFunc].sched;
}
```

Code Example 24. Chorusing Instrument with a Break-Point Envelope

The *Kbpenv* constructor message takes an envelope table, an amplitude scale value, a bias offset, a time scale value, and a completion function as its arguments. If you change the constructor above to use the *env2* function, the note will have a distinctly different envelope.

This example still does not let us change the envelope's characteristics on a note-by-note basis; to do that, we have to construct the break-point envelope function "on-the-fly" in the instrument, as we'll illustrate in the next example.

Creating Parameterized Break-point Functions

To create an envelope with variable shape (e.g., attack or decay), we create the break-point envelope at run-time for each note. To do this, we create a list with the amplitude values of the break-points and the durations of the segments between the amplitude values. For example, if we want an ASR (attack/sustain/release) envelope that starts at amplitude 0 and rises to amplitude 1 in 0.2 seconds, then decays to amplitude 0.6 at time 0.9 seconds, dropping back to amplitude 0 at time 1.0, we would have the following data values:

Amplitudes:	0	1	0.6	0
	(These start/end at 0)			
Durations:	0.1	0.7	0.2	
	(These add up to 1.0)			

Note that there will always be one more amplitude value than there are durations (because the durations are given for the transitions between two amplitude values). The general shape of this envelope is shown in the figure on the right. The list that would create this envelope shape would be something like the following.

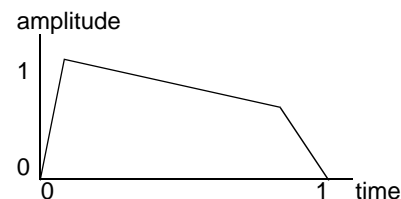


Figure 22. ASR Break-Point Envelope Function

```
[ \ignored      -- Symbolic name of the list (it's ignored)
[0 1 0.6 0]    -- Amplitudes -- First = last = 0
[0.2 0.7 0.1]  -- Segment durations -- Sum = 1.0
-1 -1]         -- -1 values for advanced users
```

This demonstrates that one can create an arbitrary envelope shape using run-time data (i.e., you do not have to pre-edit the envelope function as we had to using envelope tables and Ktransient unit generator). This kind of creation is illustrated in the instrument examples below; we assign the 2-D list to a variable and use it as the table argument to the Kbpenv unit generator constructor message.

An Instrument with Attack/Decay Control

The instrument below has variable attack and decay times. The start function plays 15 notes in succession. At the beginning, we use a triangle-style envelope (duration = 0.5 sec., attack = decay = 0.25 sec.) and the timesRepeat loop in the start function decreases both the attack and decay times with each successive note.

```

-- Single-oscillator instrument with attack/decay variables (sustain level = 0.7).
-- This uses the technique of creating the envelope "on the fly"
-- from a 2-D list of amplitude values and durations.
defaudioout L R;
deftable tabl1;                                -- Wave table for the chorusing oscillator.

start {
  var start, dur, freq, att, dec;

  start = 0;                                     -- Set up starting conditions
  dur = 0.5;                                     -- Initial start time = 0.
  freq = 200 + 40.rand;                         -- Play 2 notes per second.
  att = 0.25;                                   -- With slight frequency variation.
  dec = 0.25;                                   -- Start out with att = dec = (dur / 2)
                                              -- (i.e., a triangle envelope).

  15.timesRepeat({                               -- Play 15 notes with a dynamic envelope.
    [start, `note, dur, freq, att, dec].sched;-- Repeat this loop 15 times.
                                              -- Play a note at time start in the future.

    start = start + dur;                       -- Then increment the start time for the next note,
    att = att / 2;                             -- Halve the attack time,
    dec = dec / 1.5;                           -- Divide the decay time by 1.5,
    freq = 200 + 40.rand;                     -- And pick a new frequency for each note.
  });                                           -- End the timesRepeat({}) loop.
  [(start + dur + 0.1), `dspKill].sched-- Turn off the DSP engine when done.
}

-- Chorussing oscillator instrument with att/dec parameters
--
note {                                          -- Arguments are duration, frequency, attack, and decay.
  arg dur, freq, att, dec;
  var osc, b_points, env;                    -- Declare local variables.
  osc = Acoscili(tabl1, freq, 2);            -- Create a chorussing oscillator object.
                                              -- Create the list of [[values][durations]] for the envelope

  b_points = [\name,                          -- Name of function (symbol, ignored)
    [0, 1, 0.7, 0],                         -- Amplitudes (4 values) (sustain = 0.7)
    [att, (dur - (att + dec)), dec],         -- Durations of segments (3 segments)
    -1, -1];                                -- 2 magic "-1"s (required)

  -- Create the break-point envelope unit generator.
  -- table, amp, bias, timescale, completion function)
  env = Kbpenv(b_points, 0.25, 0, 1,         `dspRemove);

  -- Evaluate the oscillator and envelope generator in the DSP loop.
  { (osc.value *! env.value).out(L).out(R) }.dspAdd;
}

```

Code Example 25. An Instrument with Attack/Decay Parameters

Because we have only three segments in the envelope above (according to Figure 22 above) we can set the duration of the middle segment to be $(dur - (att + dec))$ as we did in the second sub-list of the 2-D list `b_points`. This method can be extended to arbitrarily complex break-point functions, and they can be used for any control parameters, as we'll illustrate in the next example.

FM With Dual Envelopes

The last example of this chapter shows a somewhat more sophisticated FM instrument that includes attack/decay parameters for both the amplitude and modulation index envelopes, a stereo position parameter, and low-frequency vibrato. The vibrato (essentially a low-frequency FM) is created using a control-rate sinusoidal oscillator (`Ksinosc`) whose value is added to the FM oscillator's frequency inside the DSP loop. The start function calls the instrument definition function with parameters that are arrived at by adding random variations to a set of base values.

-- FM instrument with 3-segment break-point envelopes for amplitude and modulation index,
-- attack and decay parameters, as well as LFO vibrato, and stereo panning (10 parameters)

```
defaudioout L, R;          -- Declare output buffers.

start {                    -- Start function plays notes on the FM instrument.
  var dur, ampl, pitch, index, ratio, pos, att, dec, i_att, i_dec;

  dur = 0.75;              -- Set up parameters
  ampl = 0.75;             -- Constant duration.
  pitch = 48 + 12.rand;     -- Constant amplitude.
                          -- Pitch is random within an octave.

  index = 1 + 5.0.rand;    -- Modulation index between 1 and 6.
  ratio = 1 + 0.1.rand;    -- C:M ratio between 1.0 and 1.1.
  pos = 1.0.rand2;         -- Position between -1 and +1.

  att = 0.2.rand;          -- Amplitude attack time between 0.0 and 0.2.
  dec = 0.1 + 0.3.rand;    -- Amplitude decay time between 0.1 and 0.4.

  i_att = 0.3.rand;        -- Index attack time between 0.0 and 0.3.
  i_dec = (dur - i_att) / 2; -- Index decay time is 50% of (duration - i_attack).

                          -- Play a note by calling the instrument definition function.
  fm_instr(dur, ampl, pitch, index, ratio, pos, att, dec, i_att, i_dec);

                          -- Repeat yourself at the end of the current note.
  [dur, thisFunc].sched;
}
```

```

-- FM instrument with ampl and mod_index attack and decay, vibrato, and stereo position.
--
fm_instr{
    -- The function has 10 arguments.
    arg dur, ampl, pitch, index, ratio, pos, att, dec, i_att, i_dec;
    -- Declare local variables.
    var osc, a_func, i_func, a_env, i_env, hz, vib, depth;

    hz = pitch.midicps;    -- Convert MIDI pitch to Hz.

    -- Create an FM oscillator (carrier_freq, mod_freq, index).
    osc = Aposcil(hz, hz*ratio, 0);
    -- Make the index envelope function using note parameters.
    i_func = [\ind, [ 0, 1, 0.8, 0 ],
              [i_att, (dur - (i_att + i_dec)), i_dec],
              -1, -1];
    -- Create the modulation index envelope generator.
    -- Arguments (func_list, ampl, bias, time-scale, complFunc).
    i_env = Kbpenv(i_func, index, 0, 1, `dspRemove);

    -- Make amplitude envelope list.
    a_func = [\amp, [ 0, 1, 0.5, 0 ],
              [att, (dur - (att + dec)), dec],
              -1, -1];
    -- Create the amplitude envelope generator.
    a_env = Kbpenv(a_func, ampl, 0, 1, `dspRemove);

    vib = Ksinosc(3 + 3.0.rand);    -- Create a low-frequency vibrato oscillator.
    -- Vibrato frequency is 3 - 6 Hz.
    depth = 12.rand;    -- Set the vibrato depth (in Hz).

    -- Apply vibrato and ampl/index envelopes in the DSP loop.
    --      freq + (vibrato * depth) mod_f   index      *   amplitude
    { (osc.value((hz + (vib.value * depth))), \,   i_env.value) *! a_env.value)
    --      pan using stereo position parameter.
    .pan2out(pos, L, R) }.dspAdd;
}

```

Code Example 26. FM Instrument with Parameterized Envelopes

This last example shows the power and also the terseness of SC programs.

5.5. Wave Table Vector Synthesis



Vector synthesis is a technique whereby one can control the mixing of a number of synchronized oscillators that have different wave forms. In the simple case, one can cross-fade between a set of wave tables within a single note, leading to a dynamic timbre over which one has straightforward control; one parameter

controls the mixing of the wave tables, and hence the instrument's timbral development over time. In SC, there is an oscillator object (*Avoscil*) that takes a list of wave tables, and an index into that list. If the index is not an integer, the oscillator mixes two of the wave forms in its list to get the output wave.

It is quite useful to either have GUI control over the index's value during a note, or to have a special index envelope that controls the timbral development of the note. These two techniques are demonstrated in the two instruments below.

Vector Synthesis with an Index Slider

The first example of vector synthesis uses a vector oscillator with a list of four very different wave tables. After defining the instrument, you can use the table editor to create the four wave tables. For this example, I've set them for maximum contrast.

-- Vector synthesis example with a table index slider in the GUI.
 -- This example uses a multi-table oscillator with interactive control over the table mixing.

```
defaudioout L, R;
deftable t1, t2, t3, t4;      -- 4 very different wave tables (look in the on-line editor)

start {
  var dur, ampl, freq, att, dec;
                                -- Set up starting parameters.
  dur = 5;                      -- Play long notes so you can hear the slider's effect.
  ampl = 1;                     -- Constant amplitude.
  freq = 48 + 12.rand;          -- Pick new pitches within an octave.
  att = 0.4.rand;               -- Random attack and decay.
  dec = 0.5.rand;
                                -- Play a note in the "wave" instrument.
  wave(dur, ampl, freq, att, dec);
                                -- Repeat notes every 5 seconds.
  [dur, thisFunc].sched;
}

-- Vector instrument with GUI control over the cross-fade between 4 waves.
--
wave {
  -- Arguments are duration, amplitude, frequency, attack, and decay.
  arg dur, ampl, freq, att, dec;

  var osc, a_func, a_env, hz, vib, depth;

  hz = freq.midicps;            -- Convert MIDI pitch to Hz
                                -- Create the vector oscillator (table list, freq, table index).
  osc = Avoscili( [t1 t2 t3 t4], hz, 0);

                                -- Make the amplitude envelope list.
  a_func = [\amplitude, [ 0, 1, 1, 0 ], [att, (dur - (att + dec)), dec], -1, -1];
```

```

-- Create the amplitude envelope generator.
a_env = Kbpenv(a_func, ampl, 0, 1, `dspRemove);

-- Define the vibrato oscillator and vibrato depth.
vib = Ksinosc(3 + 4.0.rand);
depth = 4.rand;

-- Apply vibrato, amplitude envelope, and table index GUI slider in DSP loop (slow).
--      freq      vibrato      table index      amplitude      outputs
{ (osc.value((hz + (vib.value *! depth))), 1.getItemValue) *! a_env.value) .out(L).out(R)
}.dspAdd;
}

```

Code Example 27. Vector Synthesis with Table Index Slider

This program will repeat 5-second-long notes that allow you to move the slider back and forth to hear the distinct timbres of the four wave tables, and the various hybrid timbres you get by moving the slider to intermediate positions and thereby mixing two of the wave tables together.

Vector Synthesis with Random Index Function

One can also generate dynamic timbres by applying a dynamic envelope to the vector oscillator's table index. This way, you could use one wave table for the attack part of the note only, another for the "steady-state" part, and yet another for the note's decay portion. The example below generates this envelope using a random break-point function. The table index envelope has three segments, each of which is $\text{dur}/3$ long. The y values of the break-points are each $4.\text{rand}$ (i.e., 0, 1, 2, or 3). This means that the timbre might change rapidly several times during one segment of the envelope (if the starting value of the segment is 0 and the ending value is 3, for example), leading to a dynamic and unpredictable timbral development.

```

-- Vector synthesis with a random envelope that fades between the 4 tables during a note.
--
defaudioout L, R;
deftable t1, t2, t3, t4;      -- 4 very different wave tables.

start {
  var dur, ampl, freq, att, dec;
                                -- Set up starting values.
  dur = 2;
  ampl = 1;
  freq = 48 + 12.rand;
  att = 0.4.rand;
  dec = 0.5.rand;
}

```

```

        wave(dur, ampl, freq, att, dec); -- Play a note on the wave instrument
    [dur, thisFunc].sched;           -- Repeat notes.
}

-- Vector instrument with random cross-fades between 4 wave tables.
--
wave {
    arg dur, ampl, freq, att, dec;
    var osc, a_func, i_func, a_env, i_env, hz, vib, depth;

    hz = freq.midicps;           -- Map the pitch to Hz.
                                -- Create the vector oscillator
    osc = Avoscili([t1 t2 t3 t4], hz, 0);
                                -- Make the amplitude envelope list and envelope generator.
    a_func = [amplitude, [ 0, 1, 1, 0 ], [att, (dur - (att + dec)), dec], -1, -1];
    a_env = Kbpenv(a_func, ampl, 0, 1, `dspRemove);

                                -- Make an index function that "wanders" among the 4 tables.
    i_func = [index,
              [4.rand, 4.rand, 4.rand, 4.rand]
              [dur/3, dur/3, dur/3] -1 -1];

    i_env = Kbpenv(i_func, 1, 0, 1, `dspRemove);

                                -- Define the vibrato oscillator and depth.
    vib = Ksinosc(3 + 5.0.rand);
    depth = 4.rand;

    -- Apply vibrato and ampl/index envelopes in DSP loop
    --      freq      vibrato      table index      *      amplitudemono output
    { (osc.value((hz + (vib.value *! depth))), i_env.value) *! a_env.value).out(L).out(R)
    }.dspAdd;
}

```

Code Example 28. Vector Synthesis with a Random Table Index Envelope

Vector synthesis is very powerful for instruments that should have special attack or decay characteristics such as a “chiff” in the attack (as in woodwinds) or an inharmonic final decay (as in damped strings).

Granular Synthesis Using a Sound File

The next example is the granular synthesis demonstration by James McCartney. This is one of the most fun examples for novices to experiment with, but is also quite a complex program. The GUI has sliders that determine the ranges of values used for random granulation of a stored sound file. Readers unfamiliar with granular synthesis are

referred to (Roads 1996). The program in this example reads through an on-disk sound file, granulating it with variable rate (the speed of progressing through the file with the read pointer that chooses where to start a grain), pitch (the transposition of the selected grains), and grain duration/overlap (which determine the grain density).

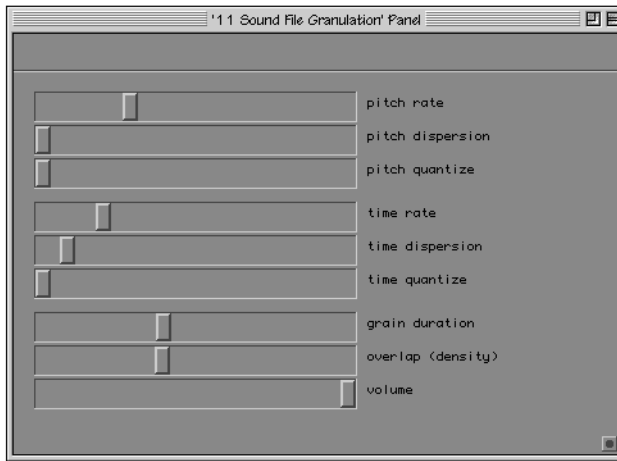


Figure 23. Sound File Granulator GUI

The first two of these parameters are controlled by sets of three sliders each that set the base value, the random range (dispersion), and the quantization step size (See the figure). Readers are encouraged to experiment (play) with this instrument, and then to analyze the program below as an exercise before moving on to the next topic.

```
-- Granular synthesis instrument that reads grains from a stored sound file.
-- 9 GUI sliders control the grain pitch (1-3), the reading rate (4-6), the grain density (7-8),
-- and the overall amplitude (9).
```

```
(***** GUI alider assignments
  slider 1 = pitch rate  0..4
  slider 2 = pitch dispersion  0..2
  slider 3 = pitch quantize  0..0.5

  slider 4 = time rate  -5..+4
  slider 5 = time dispersion
  slider 6 = time quantize

  slider 7 = grain duration  2..8
  slider 8 = overlap  -3..3
  slider 9 = amplitude  0..0.3
  *****)
```

```
defaudiobuf floating_1;          -- Input sound file (name of file = name of variable).
defaudioout L R;
```

```
init {                            -- Init function (called once at compile time).
  loadAllAudio;                   -- Locate and load the audio buffer declared above.
}
```

```

start {
    tpos = 0.0;
    overlap = 0.5;
    deltat = dur = 2.0 ** 7.getItemValue * 0.001;
    chan = [L R].choose;

    -- Instrument runs in the start function.
    -- Initialize variables (without declarations).
    -- Reading position in the file.
    -- Grain overlap.
    -- Amount to increment the reader for next grain.
    -- Choose a random output channel for each grain.

    -- Time parameters: (** means "to the power of")
    tpos = tpos + (2 ** 4.getItemValue * deltat); -- Increment the reading position.
    tdisp = (5.getItemValue ** 4).rand2; -- Read the random dispersion.
    tquant = 6.getItemValue; -- Get the quantization value.

    tpt = (tpos + tdisp).round(tquant); -- Calculate the actual reading point.

    -- Pitch parameters:
    pquant = 3.getItemValue; -- Read pitch quantization.
    pdisp = 2.getItemValue.rand2; -- Randomize the pitch dispersion.

    pch = (1.getItemValue + pdisp).round(pquant); -- Calculate the actual pitch value.

    amp = 9.getItemValue; -- Read the amplitude slider.

    -- Create the grain unit generator.
    -- file_buf offset rate dur amp completion function
    grain = Acpgain(floating_1, tpt, pch, dur, amp, `dspRemove);

    { grain.value.out(chan) }.dspAdd; -- DSP loop, play just 1 grain.

    overlap = 0.5 ** 8.getItemValue; -- Update grain parameters.
    nextdur = 2.0 ** 7.getItemValue * 0.001; -- Read overlap GUI item.
    dt1 = nextdur * (overlap - 1.0) + dur; -- Calculate duration of next grain.
    dt2 = dur * overlap; -- delta time 1 is additive overlap.
    -- delta time is duration * overlap.

    deltat = dt1 max: dt2; -- Use the maximum of the delta t estimates.

    [deltat, thisFunc].sched; -- Reschedule at deltaT in the future.
    dur = nextdur; -- Reset grain duration.
}

```

Code Example 29. Granulation of a Sound File

5.6. Real-Time Control in SuperCollider



Up to now, we have only used GUI sliders for real-time parameter input. In this chapter, we will look into the other facilities for real-time control of SC programs.

Mouse Control of Frequency

The most obvious input device for the SC programmer is the mouse. The special variables (or built-in functions if you like) `mouseX` and `mouseY` return the *x* and *y* coordinate of the mouse's current position, respectively. The origin of the geometrical plane defined by `mouseX/Y` is in the upper-left of the screen; the *x* coordinates get larger as you move to the right; the *y* values get larger (positive) as you move down.

We can add mouse control of frequency to the simple chorusing oscillator instrument as shown below (lower pitches are to the left). This instrument also has a random rhythm (random delay between notes) because we set the rescheduling delay to $(0.1 + 0.2.\text{rand})$.

```
-- Wave table osc. with random rhythm and mouse control of frequency.
--
defaudioout L R;
deftable tab1, env1;          -- Declare wave and envelope tables.

start {
    -- Chorus oscillator with mouse control over the frequency.
    osc = Acoscili(tab1, (100 + mouseX / 2), 1);

    -- Create a 1-sec. amplitude envelope (table, dur, amp. bias).
    amp = Ktransient(env1, 1, 0.2, 0, `dspRemove);

    -- Play them in the DSP loop.
    { (osc.value *! amp.value).out(L).out(R); }.dspAdd;

    -- Repeat notes every 0.1 - 0.3 sec.
    [(0.1 + 0.2.rand), thisFunc].sched;
}
```

Code Example 30. Mouse Control of an Oscillator's Frequency

You can, of course, use the mouse's two coordinates for different parameters within an instrument, such as having the *x* dimension control the frequency of an oscillator and the *y* dimension setting a filter frequency or some rhythmic property. (As an exercise, try changing the above instrument so that the `mouseY` variable sets the note repeat rate.)

The interactivity of real-time input is determined by the sound output buffer size (the Frame size). To make the response time shorter, type **Command-g** to get the Set Globals dialog box and decrease the Frame size from the default of 4096 to 1024 (and observe the increase in CPU load to to the increased sound manager overhead).

Reading Values from MIDI

One can get MIDI controller values at any time by using the functions `ctlIn(ctlr, chan)`—to get the controller value for the given controller number on the given MIDI channel—and `bendIn(chan)`—to get the pitch-bend value for the given channel. Both of these messages answer a number between 0 and 127, which your code can scale and offset to any desired range.

To make an instrument that is triggered by MIDI note-on commands, you use a different process—you register a *voicer* object for it using the `setNoteFunc` function as illustrated in the code fragment below.

```
init {          -- In your init function, create Voicer object and setup MIDI handler functions.
  var vc;
                -- The Voicer's constructor takes an instrument class name and a
                -- MIDI channel number. In this case, you need to define a class
                -- named "instrument."
  vc = Voicer('instrument', 8);

                -- The call to setNoteFunc assigns the voicer's noteon() function to
                -- handle MIDI note-on commands; this will instantiate and call your
                -- instrument.
  setNoteFunc({ arg note veloc chan;
    vc.noteon(note veloc chan);
  }, 1);        -- This means it uses MIDI channel 1.

                -- To receive continuous control data, use the setCtlFunc message;
                -- e.g., to get data from the two pre-defined pedals, use the following.
                -- Your instrument object would then need methods named
                -- sustainPedal and sostenutoPedal.
  setCtlFunc({ arg ctlNum val chan;
    if ctlNum == 64 then vc.sustainPedal(val); end
    if ctlNum == 66 then vc.sostenutoPedal(val); end
  }, 1);
}
```

Code Example 31. MIDI Voicing Outline

The next section will give a complete example of the use of Voicer objects.

MIDI + GUI Input

In this section, we will create an instrument that is triggered by MIDI note-on commands and controlled by a GUI slider for pitch transposition. Because the Voicer object creates a new instance of the instrument for each note you play, it is created with the name of an instrument *class*, so we'll create our instrument as a class rather than a function. The only difference for now is that we place the word *class* in front of the instrument definition function. If the topic of object-oriented programming (OOP) is entirely new to you, you might skip ahead and read the first few parts of the chapter on OOP in the next Part of this book.

```
-- MIDI voicing example -- Based on James McCartney's Tutorial 8 (p. 64 ff)
--
defaudioout L R;
deftable table1;      -- The oscillator wave table.
defenvelope aenv;     -- An envelope rather than a table.

init {                -- Init function sets up Voicer object.
  var vc;
                        -- Create a Voicer and setup MIDI "event handler" functions.
                        -- The Voicer takes a class name and creates instances.
  vc = Voicer(`instr1, 8); -- It uses instrument class instr1 and can play up to 8 notes.

                        -- Set up an event handler for note-on commands.
  setNoteFunc({ arg note, veloc, chan;
    vc.noteon(note, veloc, chan);
  }, 1);              -- The Voicer will map MIDI channel 1 to instr1.

                        -- Define event handlers for the MIDI pedal controls.
  setCtlFunc({ arg ctlnum, val, chan;
    if ctlnum == 64 then vc.sustainPedal(val); end
    if ctlnum == 66 then vc.sostenutoPedal(val); end
  }, 1);              -- Map channel 1 pedal to instr1 envelopes.
}

-- NB: There is no start function! (It's all triggered from MIDI.)

-- Instrument class -- instance objects are created automatically by the voicer.
--
class instr1 {
  arg key, veloc, chan, voicerobj;
                        -- This (anonymous) method is the class constructor;
                        -- its arguments are the class's instance variables.
  var freq, amp, osc, aenv;
                        -- Scale the frequency and amplitude inputs.
  freq = key.midicps;   -- MIDI key number translation.
  amp = veloc/128 ** 2 * 0.55; -- MIDI key velocity scaling.
```

```

-- Create an interpolating oscillator that uses table1.
osc = Aoscili(table1, freq + 1.getItemValue);

-- Create a break-point envelope.
-- (Please note the unusual completion function.)
aenv = Kbpenv(aev, amp, 0, 1, { dspRemove; voicerobj.remove(this) });

-- DSP loop.
{ (osc.value *! aenv.value).out(L).out(R) }.dspAdd;

-- These two class methods are required by the Voicer.
method release { aenv.release; }
method steal { aenv.steal; }
}

```

Code Example 32. MIDI Voicer Triggering an Instrument

In this example, we set up the Voicer in the `init` function above, and declared an instrument class for the synthesis portion of the program. The only important changes to the instrument definition are that it takes the Voicer as an argument (instance variable) and that the envelope's completion function is a closure that calls `dspRemove` and then sends the `remove` message to the Voicer to delete the instrument instance entirely. The final two methods in the instrument class are required by the Voicer so that it can handle note-off commands and note stealing.

There are obviously many more possibilities for MIDI control of SC programs, but this must suffice for our introductory tutorial.

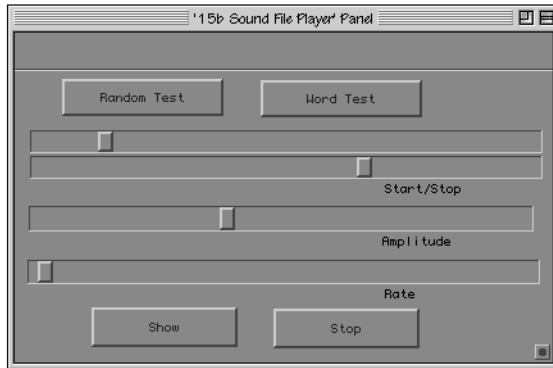
5.7. Sound File Player Instruments

Up to this point, we have only dealt with synthetic (oscillator-generated) sound. SC also provides easy methods of processing recorded or even live sound, as we'll investigate in the following chapters.



One common technique is to read samples from on-disk sound files, and mix or process them in SC programs, sending the output to the sound manager in real-time. In some cases, you may want to play entire sound files with some pitch or time scaling—as with a sampler or vocoder—and in other cases it is useful to be able to play several different (possibly overlapping) segments from a sound file that is longer than one “note.” Both of these modes of operation are easy in SC using the audio buffer reader object `Abufrd`.

The instrument I present here reads samples from a sound file, and allows you to select the start/stop points (cue points) for each note.



I use a file that contains a spoken sentence—"Shu gu dwan ren shing," the first line of the famous T'ang dynasty Chinese poem Ywe ye, yi jr di by Tu Fu (it is read by Sinologist Ernest Chen). The GUI for the instrument (shown on the left) allows you to vary the start/stop points, to run the two built-in demo functions, and to print the current start/stop times to the transcript view.

Figure 24. Sound File Player GUI

The program is structured so that the `init` function loads the sound file into the audio buffer object, and the `start` function starts the GUI demo routine. The other demo functions can be called using buttons in the GUI window.

The main instrument is called `sndFile` and is a general-purpose function that takes as its arguments the file buffer (a variable defined using `defaudiobuf`), start offset (where in the file to begin reading, in seconds), the amplitude scale, the note duration, the stereo position, the read rate (set it to 1 to read through the sample at the default sample rate), and the attack and decay times for a trapezoidal envelope (i.e., an ASR envelope with a sustain level = 1.0).

The program also defines a "cue section list" called `words`. This is a data list where the first element is a list of the words in the sentence (stored as symbols), and the second element is a list of lists giving the start offsets and durations of each of the words in the sentence. The `playWord` example function demonstrates the use of the cue list, accessing is like a dictionary keyed by word.

```
-- Sound file player instrument with cue points and built-in demo functions.
--
defaudioout L R;
defaudiobuf sgdrs;           -- Input sound file variable (named after the sentence I use).
                               -- (The file name and thevariable name can be different.)

var words;                   -- Variable for cue point list.

const length = 1.44;         -- Length of the sound file.
```

```

init {
    -- Init function (called at compile time).
    loadAllAudio(sgrds, "sgdrs");    -- Locate and load the audio buffer declared above
    -- given a variable name and a file name.

    -- Define the cue list (a 2-D array); the format is [[name]
    -- [start, dur]]. You can treat this like a dictionary (see below).
    words = [[\shu \gu \dwan \ren \shing]
              [[0.03, 0.29] [0.32, 0.26] [0.63, 0.23] [0.83, 0.26] [1.09, 0.34]]];
}

start {
    -- The start function runs the interactive GUI-based demo.
    playGUITest;
}

-- Sound file player instrument -- takes as arguments the buffer, start, amp, dur, etc.
--
sndFile {
    arg file, offset, amp, dur, rate, pos, att, dec; -- Function arguments.

    var snd, list, env;
    -- Local variables.

    -- Create a trapezoidal break-point envelope list.
    list = [ \envelope [ 0 1 1 0 ] [ att, (dur - (att + dec)), dec] -1 -1 ];

    -- Create the audio buffer reader object given the
    snd = Abufdr(file, offset, rate); -- input buffer, the start offset, and the reading rate.

    -- Create the amplitude envelope.
    env = Kbpenv(list, amp, 0, 1, `dspRemove);

    -- DSP loop takes file reader and envelope...
    { (snd.value *! env.value).pan2out(pos, L, R) }.dspAdd;
}

-- Play the given word (selected by a name that is used as a key in the "word dictionary").
--
playWord {
    arg name, amp, pos;
    var data;
    -- Arguments are word, amplitude, position.
    -- data is the start/duration sub-list.

    name.post;
    data = words.assocAt(name);
    -- Display the name.
    -- Get the start/duration times by using the list like a
    -- dictionary (e.g., accessing it by word).

    -- Call the sound file player instrument.
    -- buffer offset amp duration rate pos att dec
    sndFile(sgrds, (data @ 0), amp, (data @ 1), 1.0, pos, 0.01, 0.02);
}

```

```

-- Test and Demo Functions
--

-- Rand test -- Play 20 notes with start offset in the sound buffer selected at random.
--
playRandTest {
    var file, offset, dur, amp, rate, pos, att dec;

    file = sgdrs;                -- These don't change -- buffer name.
    dur = 0.15;                  -- Note duration.
    rate = 1.0;                  -- Reading rate (no transposition).

    for i = 0; i < 20; i = i + 1; do -- Loop 1 to 20
        offset = 1.2.linran;      -- Use a linear distribution to get the starting offset.
        amp = 0.2 + 0.6.rand;     -- Some variation in amplitude.
        pos = 0.5.rand2;         -- Random position near the stereo center.
        att = 0.001 + 0.005.rand; -- Short attack.
        dec = 0.05 + 0.15.rand;   -- Longer decay.

        -- Schedule a note on the sndFile instrument.
        [(i * dur), `sndFile, file, offset, amp, dur, rate, pos, att, dec].sched;

    end.for -- End of the for loop.
}

-- Word test -- Play 50 words from the pre-defined list of cue points such that you start at the
-- first word and end at the last word, but have some random "jitter" as you step through.
--
playWordTest {
    var count, list, dur;        -- Local variables.

    count = 50;                  -- Play 50 notes.
    list = [\shu \gu \dwan \ren \shing]; -- Taken from this list of words.
    dur = 0.08;                  -- With a constant repetition rate.

    for i = 0; i < count; i = i + 1; do -- For loop with 'count' repetitions
        index = (((i * 5.0) / count) + 0.7.rand2); -- This goes from 0 to 5 during the
        -- loop, but with a bit of randomness

        index.post;              -- Post the index to the transcript.

        -- Now play a note.
        -- start instrument word amp pos
        [(i * dur), `playWord, (list |@| index), (0.2 + 0.6.rand), 0.6.rand2].sched;

    end.for -- end the loop.
}

```

```

-- GUI test -- This reads the slider values and repeats a cue section to allow you to pick
-- out a region. If you then press the Show button, it will display the start/stop times in
-- the transcript view.
--
playGUITest {
    var offset, stop, amp, rate, pos;

    offset = 6.getItemValue * length;-- Get the offset slider.
    stop = 8.getItemValue * length; -- Get the stop time slider.

    if (stop <= offset) then          -- Catch it if the users tried to drag offset > stop
        stop = (offset + 0.2) min: length;-- Reset the value.
        8.setItemValue(stop / length);    -- And change the slider.
    end.if

    amp = 1.getItemValue;           -- Get the amplitude slider value.
    rate = 1 / (2.getItemValue);    -- The rate is the inverse of the slider value.
    pos = 0.5.rand2;                 -- Random position near the center of setereo.

    -- Now play the note on the sndFile instrument.
    sndFile(sgdrs, offset, amp, (stop - offset), 1.0, pos, 0.01, 0.02);
    -- Repeat the function at rate rate.
    [rate, thisFunc].sched;
}

-- Show function -- Print the sample start/stop times in the transcript
--
showGUI {
    ['Start: ', (6.getItemValue * length),
    'Stop: ', (8.getItemValue * length)].post;
}

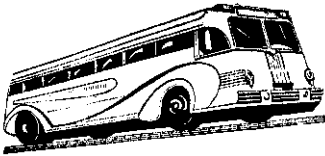
```

Code Example 33. Sound File Player Instrument

If you work with concrete sounds, there are many possible extensions or modifications to this kind of program that might be of interest. In real life, I use versions of the sound file player functions given above that take the name of the cue point (like play-Word in the above code) as an argument, and that also support pitch shifting and time stretching. These extensions are left as an exercise to the reader.

Other extensions might be more sampler-like, whereby you would have a known base pitch for your sample and provide the ability to play notes with a given pitch.

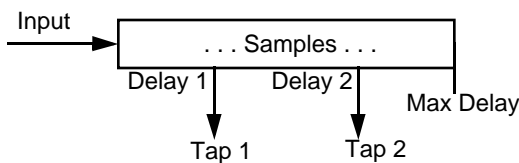
5.8. Signal Processing



All of the instruments we have used to this point have been based on some sort of synthesis or stored sound (even in the case that the function was granulating a file from the disk). In this chapter, we will look into signal processing of live inputs within SC programs.

Multi-stage Processing for Reverb

In this section, we will write a program that consists of a repeating note instrument and a basic echo effect processing function. This example uses a delay line, which is simply a sample buffer that can be written into like an output (using the out message), but which can also have one or more reader “taps” at various delay times.



One can visualize this as in the figure to the left, which shows the input signal feeding into the sample buffer (whose length is given by Max Delay). After delay time 1, the signal is sent to tap 1 output, and after delay 2, it is sent out tap 2.

Figure 25. Structure of a Delay Line

The delay times can change over time (“moving taps”), and there is no practical limit to the number of taps allowed on a delay line.

The instrument function in the code example below is a repeating pulse instrument in which the frequency is random and the repetition rate is based on the mouse’s x coordinate. The output of this instrument is sent to the output channels L and R, and also written into a delay line in DSP stage 1. Another instrument (called *fx* and put in DSP stage 3) reads from the delay line and writes to the outputs and back into the delay line (feedback). This has an echo-like effect, as you can hear by experimenting with it.

-- Using a delay line and multiple DSP stages. (See manual p. 58.)

--

```
defaudioout L R;
deftable tab1 env1;
defdelay dly1(0.4);
```

-- Declare wave form and envelope tables.

-- Create a 0.4 second delay line called dly.

```
start {
  instrument1;
  fx;
}
```

-- The synthesizer and processor functions run in parallel.

```

-- Chorusing oscillator instrument that repeats with rate determined by mouseX.
--
instrument1 {
    rate = mouseX / 200 + 0.1;
    -- Repeating notes with repetition rate based on mouseX.
    -- Chorusing oscillator with frequency from 100 to 900 Hz.
    osc = Acoscili(tabl1, (100 + 800.rand), 1);
    -- Create an envelope generator.
    amp = Ktransient(env1, rate, -18.dbamp, 0, `dspRemove);

    -- Write samples to the output and delay line in DSP stage 1.
    { (osc.value *! amp.value).out(L).out(R).out(dly1) }.dspAdd(1);

    [rate/2, thisFunc].sched;-- Have a 50% overlap between notes.
}

-- Recirculating delay line "echo" effect (at DSP stage 3).
--
fx {
    -- Read samples from the delay line (at 0.2 seconds delay),
    -- scale them, and write them out (as well as back into the
    -- delay line) in DSP stage 3.
    { (tap(dly1, 0.2) *! 0.8).out(L).out(R).out(dly1) }.dspAdd(3);
}

```

Code Example 34. Tapped Delay Line Instrument

This program uses two different stages of the DSP loop by adding the first instrument to stage 1 and the effects processor to stage 3 (look at the arguments of the `dspAdd` messages). The rule is that for each sample buffer, all tasks in any stage of the DSP loop will be completed before the next stage is processed. Therefore, we can be sure that the samples we write into the delay line in the instrument will go there before the effects instrument starts to read from the delay line.

Better-sounding reverberators can be written using SC's delay line and comb filter functions (coming below).

Using Audio Input

The example below illustrates the processing of a real-time sound input. The first part demonstrates a simple "echo" function that reads the real-time audio inputs and plays the data right back out the output. We define stereo audio inputs using the `defaudioin` keyword and read from them using the `in` message.

Note that because of the default size of 4096 samples in the sound manager I/O queue, there will be a delay of at least 0.2 seconds between the input and output sounds. You can experiment with making the output buffer (Frame) size smaller and hear the difference and see the increase in computational overhead.

```

-- Basic Audio Signal Processing Example -- Echo Program
-- This simply reads input and plays it back out the outputs.
--
defaudioin Lin Rin;          -- Two inputs.
defaudioout L R;             -- Two outputs.

start { {
    in(Lin).out(L);          -- Read left-in; write left-out
    in(Rin).out(R);          -- Read right-in; write right-out.
    }.dspAdd;
}

```

Code Example 35. Audio “Echo” Program

A somewhat more interesting audio signal processing program is the delay-line-based *flanger* given in the next example. We use a delay line (declared with `defdelay` and given the maximum delay time of 0.5 seconds) and a tap that reads from it. The difference between this and the initial delay line example from above is that we move the tap delay time around with a low-frequency oscillator (LFO), and do not feed the delay line output back into it. Because the output buffer receives both the original and the delayed signal, and the delay time is changing over time, you will hear the typical comb-filter-like flanging sound.

```

-- Audio Signal Processing With Delay Lines -- A "Flanger"
-- This reads the inputs and mixes current and delayed versions to the outputs.
-- The delay time is controlled by a 0.3 Hz oscillator, leading to a "flanging" effect.
--
defaudioin Lin Rin;          -- Two inputs.
defaudioout L R;             -- Two outputs.
defdelay dLine (0.5);        -- 0.5 second delay line.

start {
    var input, lfo, delay;

    lfo = Psinosc(0.3, 0);    -- Create a 0.3 Hz polled sine oscillator
                                -- DSP loop.
    {
        input = (Lin.in +! Rin.in) *! 0.25; -- Read both inputs, add them, and scale.
        input.out(L).out(R); -- Play the in back out.
        input.out(dLine);    -- Write samples to the delay line
        dTime = (lfo.value * 0.1) + 0.1;    -- Calculate the delay time (0 to 0.2 seconds).
        tapi(dLine, dTime).out(L).out(R); -- Tap in to the delay line and write the
                                -- delayed signal to the output buffers.
    }.dspAdd;
}

```

Code Example 36. “Flanger” that Reads Live Input

Reverberation

The instrument below illustrates a more complex signal processor—a configurable reverberator built using delay lines and comb filters. This model is based on the Moorer/Loy design (see the *lprev* program in Moore and Loy 1983), whereby a multi-tap delay line (or finite impulse response [FIR] reverberator) is followed by a bank of comb filters (or infinite impulse response [IIR] reverberators) in a structure like that shown in the figure on the right.

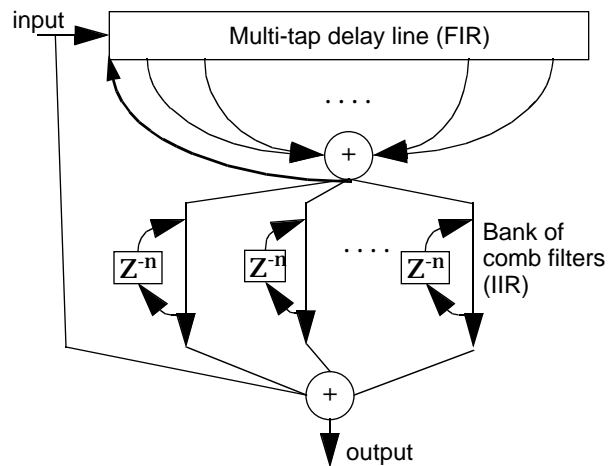


Figure 26. Moorer/Loy Reverberator Structure

This reverberator design uses the delay line to model the early sound reflections in a room, and the comb filter bank to model the more diffuse later reflections. This version of it is configurable in that the tap points and their weights and the delays and weights of the comb filters are held in tables that can be changed for each type of input sound and desired reverberation characteristic.

```
-- Moorer/Loy Reverberator
-- This program implements a classical configurable reverberator that consists of a tapped
-- delay line with up to 10 taps followed by a bank of up to 6 comb filters.
-- The sample configuration data is taken from Gareth Loy's "lprev" program that is part
-- of the UCSD CARL software distribution (Moore and Loy 1983).
--
defaudioout L R;          -- Define audio outputs.
defaudioin Lin Rin;       -- Define audio inputs.
defdelay dline(0.1);      -- 0.1-sec delay line for the initial reflections.
-- 6 delays for comb filters
defdelay c0(0.1), c1(0.1), c2(0.1), c3(0.1), c4(0.1), c5(0.1);
defbus bus;               -- Define an internal signal bus.

start {
    var tapData,           -- Early reflection tap data (time, level)
        combData,         -- Table of comb data (time level)
        combs,            -- List of comb objects
        G,                -- Global delay time
```

```
        mix,          -- Wet/dry ratio.
        tabScale      -- Tap amplitude scale.
        combScale;    -- Comb amplitude scale.

-- Initialize the tap table for early reflections
--
tapData =
  -- delay  amplitude
  [[0.0043  0.841]
   [0.0215  0.504]
   [0.0268  0.379]
   [0.0298  0.346]
   [0.0485  0.272]
   [0.0572  0.192]
   [0.0595  0.217]
   [0.0708  0.181]
   [0.0741  0.142]
   [0.0797  0.134]];

-- Initialize comb table for longer reverberations
--
combData =
  -- delay  amplitude
  [[0.050  0.46]
   [0.056  0.48]
   [0.061  0.50]
   [0.068  0.52]
   [0.072  0.53]
   [0.078  0.55]];

G = 0.99;          -- Global time ~= 3 sec.
mix = 0.5;         -- Wet/dry balance.

-- Create a list of combs
combs = [];
item = combData @ 0;
combs.add(Acombdly(c0, (item @ 0), (item @ 1) * G));
item = combData @ 1;
combs.add(Acombdly(c1, (item @ 0), (item @ 1) * G));
item = combData @ 2;
combs.add(Acombdly(c2, (item @ 0), (item @ 1) * G));
item = combData @ 3;
combs.add(Acombdly(c3, (item @ 0), (item @ 1) * G));
item = combData @ 4;
combs.add(Acombdly(c4, (item @ 0), (item @ 1) * G));
item = combData @ 5;
combs.add(Acombdly(c5, (item @ 0), (item @ 1) * G));

tapScale = mix / tapData.size;
combScale = mix / combs.size;
```

```

{
    -- DSP Loop
    input = (Lin.in +! Rin.in) *! 0.5;-- Read both inputs, add them, and scale the signal.
    input.out(dline);-- Write samples to the delay line
    (input *! mix).out(L).out(R); -- Play the in back out.

    tapData.forEach ({ -- Loop through the tap table.
        arg list;
        -- Get tap signal and scale by tap level.
        z = tap(dline, (list @ 0)) *! ((list @ 1) / tapScale);
        z.out(bus); -- Send tap signal to internal bus.
    });

    bus.out(dLine) -- Feed back on delay line.

    combs.forEach({ -- Loop through the comb table.
        arg comb;
        -- Write bus value to comb, and send output out.
        (comb.value(bus.in) *! combScale).out(L).out(R);
    });

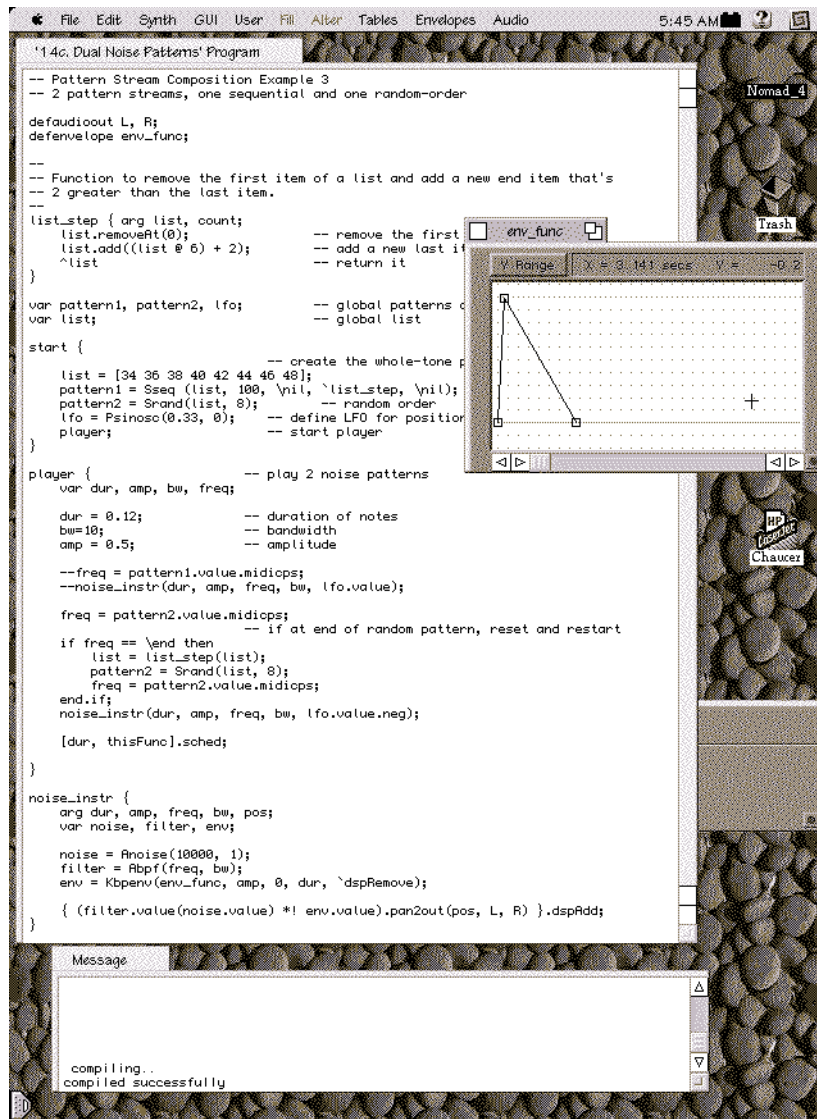
    }.dspAdd;
}

```

Code Example 37. Parameterizable Live Input Reverberator

You can also of course write versions of this kind of reverberator that read sound files from disk (e.g., created in previous passes with SC or any other synthesis/processing software). The above examples serve to introduce live signal processing with SC; one can also use filters and other kinds of processes. The on-line effects processing examples distributed with the program demonstrate many more advanced techniques.

In this part of the tutorial, we have surveyed the basic SC programming techniques that are important for day-to-day usage. The next part will look at a couple of more advanced programming techniques.



Part 6. Advanced Techniques and Libraries

In the previous Part, we surveyed several essential SC programming techniques. The next Part will present several areas of more advanced usage, including reading and writing text files, instruments with preset memory, and object-oriented programming.

6.1. File I/O

Unlike most standard MusicN languages, SuperCollider provides a collection of built-in functions for reading and writing text files. The next couple of chapters demonstrate the use of these functions to read score data from files and to save and replay parameter settings as “presets” for use in performance.



In the following examples, I have switched to C++-style function names (using embedded underscore rather than embedded upper-case letters) just so you can get the taste of it (I prefer Smalltalk-style naming in general).

Reading a Score from a File

It is often convenient to store instrument definitions and score data in separate files, especially if several scores use the same instruments. This example reuses the same FM instrument we’ve seen in the previous chapters, but reads its score from a file. When you run it, it will prompt you for the name of a score file. The file “score.sc” is included with the example code that accompanies this book. The format of the file for this example is as shown below. You can, of course, change this format if you change the file reading functions to be able to read the new format.

```
# Score for FM Instrument.
# This is a comment.

# FM instrument note commands
# Parameters:
# start instr  dur freq index  ratio
#
0.00 'fm_instr' 0.25 54 1    1
0.25 'fm_instr' 0.25 55 1    1
0.50 'fm_instr' 0.25 56 1    1
0.75 'fm_instr' 0.25 57 1    1.21

end
```

Code Example 38. Score File Format Example

In this score file format, the instrument name must be in single quotes (i.e., a symbol). There are no special statement terminator characters required, but note commands must be one-per-line. The program that reads and performs this (without including the FM instrument) is as follows.

```
-- FM instrument with a score that's read in from a file.
--
defaudioout L, R;
deftable tabl1, env1, env2;

start {
  read_score;          -- The start function calls the score reading function.
}

-- Read a score file and play note commands found there.
-- The score can include notes for several instruments.
--
read_score {
  var name, fileID, line, start, instr, args;

  line = [];            -- Create an empty list to read lines from the file into.
                        -- Get a file name by prompting the user.
                        -- Arguments are (default name, prompt string).
  name = getStringFromUser("score.sc", "Select a score.");

  name.post;            -- Post the name to the transcript.
  fileID = fopen(name, "r"); -- Open the file for reading.

  line = freadlist(fileID); -- Read the first line
  do
    line.post;          -- Loop until you see the end of the score
                        -- Post the line (for debugging).
                        -- If it's a legal command line,
    if (size(line) == 6) && ((line @ 1) != "#") then
      -- Do a multiple assignment from the list you read in.
      -- "Peel off" the first 2 items; leave the rest in the
      -- "args" list.
      # start instr ... args = line;
      -- Schedule the note at the given time and converting
      -- the name from a symbol to a function reference.
      [start, (instr.resolveName), args].sched;
    end.if;            -- End of the if (ignore all other lines).

    line = freadlist(fileID); -- Read the next line.
  end.do until (line == 0); -- End of the do; repeat until freadlist answers 0.

  fclose(fileID);      -- Close the input file.
  "Done".post
}
```

```

fm_instr{          -- The FM instrument must be changed to accept a list of arguments
  arg args;
  var duration, freq, index, ratio, osc, i_env, a_env, hz;

                                -- Read the argument list into the parameters
                                -- with a multiple assignment.
  # duration freq index ratio = args;

  ...The simple FM instrument from the last Part goes here...

}

```

Code Example 39. Score Reader Functions

You can rewrite this function to read other score formats, or to handle scores that have other types of data in them (e.g., break-point envelope functions). The scheduler has a hard limit of 512 events that can be scheduled in the future, so if you want to use longer scores, you'll have to make a version of this reading function that pauses when it gets too far ahead of event performance.

6.2. Presets and Performance Configuration

For interactive performance, it might be useful to have an instrument that has a SC GUI, but that can store and recall settings of the GUI slider values. The next few examples show you how to do this using instruments we have already introduced.

Presets and GUI Interaction

The following examples demonstrate how one can record and play back GUI interactions. In the first part, we extend the granular synthesis example introduced in the last part of the tutorial to add GUI buttons that save and restore slider settings. The simplest version has 3 buttons—*save*, *recall*, and *reset*. The *save* button reads the values of the nine sliders and stores them in a list of nine numbers (named *params*). Pressing the *recall* button simply sets these stored values back to the sliders. The *reset* button sets up “default” parameters that are given in the program code. Note that we have also added a stop button to the GUI.

```

-- Granular sound file processor with simple parameter store/recall/reset.
--
defaudiobuf sound;
defaudioout L R;

init {
  loadAudio(sound, "1.sndd");          -- load the named sound file
}

```

```

start {                                     -- Play instr1 from the start function.
    instr1;
}

instr1 {
    .. .sound file granulation instrument from above..
}

-- Functions to store and recall parameter settings.
-- These are called from the GUI buttons.
--
var params;                                -- This is a list of 9 numbers for the 9 sliders

-- Store function to read GUI slider values.
--
store_params {                             -- Store slider settings into the params[] list.
    \STORE.post;                           -- Print a symbol to the transcript.
    params = [];                          -- Declare a list for parameter values.

    for i=1; i<=9; i=i+1; do              -- Loop from 1 through 9.
        params.add(i.getItemValue);      -- Store GUI item values into the list.
    end.for;

    params.post                            -- Post the list to the transcript view.
}

-- Recall function to set the 9 GUI sliders to their stored values.
--
recall_params {                            -- Recall slider settings from params[]

    if (params = \nil) then               -- Test to make sure we've stored some parameters.
        "ERROR: You must use store before using recall.".post
    else
        for i=1; i<=9; i=i+1; do
            i.setItemValue(params @ (i - 1));-- Set GUI item values from the list.
        end.for;
    end.if;
}

-- Reset function to restore "standard" parameter settings (which are hard-coded below).
--
reset_params {                             -- Reset sliders to "default" values
    var defaults;
    defaults = [ 1 0 0 -1 0 0 3 1 1];-- These are the "default" parameters.
    \RESET.post;

    for i=1; i<=9; i=i+1; do
        i.setItemValue(defaults @ (i - 1));
    end.for
}

```

```
-- Stop Button Function
--
stop_play {                                -- Stop DSP loop with a button

    \STOP.post;
    dspKill(1)
}
```

Code Example 40. GUI Slider save/restore/reset Functions

You can easily adapt the functions above to read and store the GUI item values of any instrument you design. For the case that the GUI item values you want are not contiguous item numbers (as above, where the sliders are items 1 - 9), you can use a list with the GUI item numbers, and send it the message `collect`. (Try this as an exercise with the FM instrument; it does not have contiguous GUI slider numbers.)

Storing Presets to a File

In the next version of this program, we store multiple sets of slider configurations, using `params[]` as a 2-D list of lists. When you hit the *stop* button, these are saved to a file, one list per line.

```
-- Granular sound file processor with parameter store/recall/reset.
-- This version stores settings to a file when you stop playing.

defaudiobuf sound;
defaudioout L R;

init {                                -- load named sound file
    loadAudio(sound, "1.sndd");
}

var params;                            -- This is a list OF LISTS of 9 numbers

start {                                -- Start plays instr1.
    params = [];                        -- List of lists for parameter values
    instr1;
}

instr1 {
    ...sound file granulation instrument from above.
}

-- Functions to store and recall parameter settings (sent by GUI buttons).
--
```

```

-- Store a set of parameter values into one entry in the params[[]] list.
--
store_params {
    var setting;                                -- The current settings.

    setting = [];                                -- Make a new empty list.
    for i=1; i<=9; i=i+1; do
        setting.add(i.getItemValue); -- Add the GUI item values to the settings list.
    end.for;
    params.add(setting);                        -- Add the setting list to the 2-D params list.
    setting.post                                -- Post settings.
}

-- Stop_play: stop the DSP and store params[[]] to a file.
--
stop_play {                                    -- Stop DSP loop with a button.
    dspKill(1);
    save_params;                                -- Save the presets after stopping.
}

-- Save: Prompt the user for a file name and save the 2-D list of parameters to a text file.
--
save_params {
    var name, fileID;                            -- File name and file ID.
                                                -- Get the file name from the user.
    name = getStringFromUser("gran.params", "Select a parameter file name.");
    name.post;                                    -- Echo the name in the transcript.
    fileID = fopen(name, "w");                    -- Open the file for writing.

    params.forEach(                             -- Loop through the param list writing each sublist
        { arg elem, ind;                         -- to a new line of the output file.
            elem.fwritelist(fileID) });
    fclose(fileID);                             -- Close the output file.
}

```

Code Example 41. Functions to Save Parameter Settings to a Text File

To use this, start the instrument playing and adjust the GUI sliders to a setting you like, then press the *save* button, now adjust them to another setting, and press *save* again. After several iterations of this, you're ready to stop the instrument and save your "performance" to a file. The result of this is a text file with the list of parameter settings you stored while using the instrument. You can edit this file with any standard text editor, but be careful not to add or delete data items from the lines of the file.

Playing back Stored Presets

The last extension allows us to read in a list of parameter “presets” from the file we just created and step through them. The functions below load the parameter file on start-up, and provide you with GUI buttons that advance you to the next file setting during “playback.” The reset button restores the “standard” settings that are coded in the `reset_params` function shown above.

```
-- Granular sound file processor with parameter recall/reset.
-- This version reads settings from a file.
defaudiobuf sound;
defaudioout L R;

init {
    loadAudio(sound, "1.sndd");
}

var params;
var i;

start {
    i = 0;
    load_params;
    instr1;
}

instr1 {
    ...sound file granulation instrument used above.
}

-- Functions to load and recall parameter settings
--
-- Load a 2-D list of parameter settings from a file
--
load_params {
    var name, fileID, list;

    name = getStringFromUser("gran.params", "Select a parameter file name.");
    name.post;
    fileID = fopen(name, "r");
    params = [];

    list = freadlist(fileID);
    while (list.size != 0) do
        list.post;
        params.add(list);
        list = freadlist(fileID);
    end.while;

    -- Prompt the user for a file name.
    -- Open file for reading.
    -- Make an empty parameter list.
    -- Read the first line.
    -- Read until the end of the file.
    -- Print the line to the transcript for debugging.
    -- Add the list just read to params[] list.
    -- Read the next line.
```

```

        fclose(fileID);                -- Close the input file.
    }

    -- Recall parameter settings one-at-a-time.
    --
    recall_params {
        var setting;                    -- Setting is a list or parameter values.

        setting = params @@ i;          -- Get next preset (wrapping around).
        for j=1; j<=9; j=j+1; do        -- Loop through the GUI item values.
            j.setItemValue(setting @ (j - 1)); -- Set the GUI items from the list.
                                           -- NB: GUI item numbering is 1-based but list
                                           -- element are 0-based, hence the (j - 1).

        end.for;
        i = i + 1;                      -- Increment the preset counter.
        setting.post                     -- Print it for debugging.
    }

```

Code Example 42. Playing Back Presets from a File

Using variations of the functions given above, you can create flexible real-time performance instruments that involve any of SC's facilities: synthesis, stored sound processing, or live sound I/O.

6.3. Object-Oriented Programming in SuperCollider¹



This chapter presents an extended example of object-oriented programming (OOP or O-O technology) in SC. OOP is a powerful programming technique that allows you to structure your programs for better readability and maintenance, closer models of musical domains, and greater ease of extension.

There are several features that are generally recognized as constituting an O-O technology: *encapsulation*, *inheritance*, and *polymorphism* are the most frequently cited. We will define each of these in the sections below.

Encapsulation

Every generation of software technology has had its own manner of packaging software components—either into jobs or modules or other abstractions for groups of functions and data elements. The reasons for modular structure are to encourage a level of

1. This text is derived from the invited article “Music Representation Using Objects” that appeared in *Organised Sound* 1(1): 55-68, and is reprinted in Roads, C., S. Pope, G. De Poli, and A. Piccialli, eds. 1997. *The Musical Signal*. Lisse, The Netherlands: Swets & Zeitlinger.

privacy of data and functions (i.e., so that program components cannot depend on the internal details of other program components), to support reuse (i.e., so that functions can be shared by several programs), and to separate out a program’s interface specification (what does it do) from its implementation (how does it do it).

In traditional modular or structured technology, a module includes one or more—public or private—data types and also the—public or private—functions related to these data items. In large systems, the number of functions and the “visibility” of data types tended to be large, leading to problems with managing function names—which are required to be unique in most structured languages.

Object-oriented software is based on the concept of *object encapsulation* whereby every data type is strongly associated with the functions that operate on it. There is never a “stand-alone” data element or an “unbound” function; data and operations—state and behavior in O-O terms—are always encapsulated together.

A data type—known as a *class*, of which individual objects are *instances*—has a definition of what data storage (state) is included in the instances (called “instance variables” in the Smalltalk languages), and of what functions (behavior or “methods”) operate on instances of the class. Strict encapsulation means that the internal state of an object is completely invisible to the outside and is accessible only through its behaviors.

Any object must be sent messages to determine its state, for example, an object that represents a 2-D point in Cartesian geometry would have *x* and *y* instance variables and methods named *getX* and *getY* for accessing them. The advantage of separating the private state from the public behavior is that another kind of point might store its data in polar coordinates, with its angle and magnitude. It would be very good to be able to use these two points interchangeably, which is possible if I am only concerned with “what” they can do, and not with “how” they do it. Behaviorally, they are identical (ignoring performance for now). The figure on the left shows two kinds of point objects.

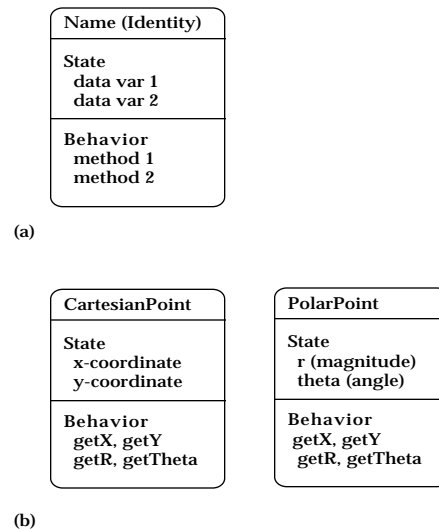
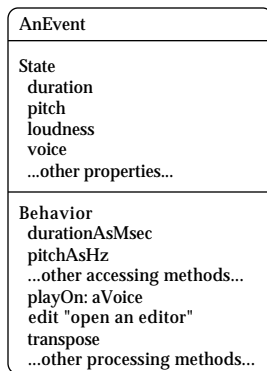


Figure 27. (a) Object encapsulation of private state and public behavior, (b) two different kinds of geometric points

As an example from the musical domain, imagine an object that represents a musical event or “note.” This object would have internal (strictly private) data to store its parameters—possibly duration, pitch, loudness, timbre, and other properties—and would have methods for accessing these data and for “performing” itself on some medium such as a MIDI channel or note list file.



Because the internal data of the object is strictly hidden (behind a behavioral interface), one can only access its state (instance variables) via messages (behaviors), so that, if the note object understood messages for several kinds of pitch—e.g., `pitchInHz`, `pitchAsNoteNumber`, and `pitchAsNoteName`—then the user would have no way of determining how exactly the pitch was stored within the note. In old-fashioned structured software technology terms, this is called “information hiding” or the “separation of the specification (the what) from the implementation (the how).” Figure 28 illustrates a possible note event object.

Figure 28. A musical event or “note” object showing its state and behavior

An example of encapsulation that we have already seen is the use of an instrument class name by MIDI Voicer objects. When you create a Voicer, you give it the name of an instrument class, and it creates a new instance of that instrument class for each active MIDI note.

Inheritance

Inheritance is a simple principle whereby classes can be defined as refinements of specializations of other classes, so that a collection of classes can form a tree-like “specialization” or “inheritance” hierarchy. At each level of this kind of subclass-superclass hierarchy, a class only needs to define how it differs from its superclass.

Examples of class hierarchies are well-known (and widely misunderstood) in the literature on O-O systems. The hierarchy of classes used to represent numbers makes a good starting example. If one had a class for objects that represented “units of measure” (often called “magnitudes” in the abstract), then it is easy to see that numbers would be one possible subclass (refinement) of it, and that the various types of numbers (integers, floating-point numbers, and fractions) might be further subclasses of the number class. There are also other possible magnitude classes that are not numbers, dates and time-of-day objects, for example. The figure below shows a possible hierarchy of magnitudes and numbers.

A musical example of this would be a system with several types of event parameters—different notations for pitches, durations, loudness values, etc. If there was a high-level (“abstract”) class for representing musical pitches, then several possible pitch notations could be constructed as subclasses (“concrete classes”) of it. Further examples of this kind of subclass-superclass hierarchy will be presented below. Remember that at each level, we are most interested in behavioral differences (rather than storage representations).

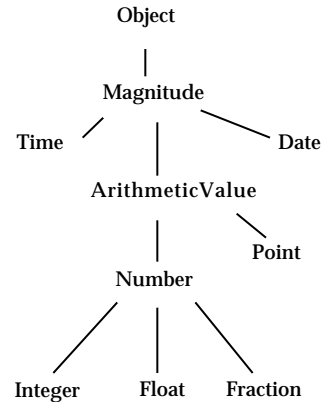


Figure 29. A class hierarchy for magnitudes

Polymorphism

In simple terms, polymorphism (also called overloading) means being able to use the same function name with different types of arguments to evoke different behaviors. Most traditional programming languages allow for some polymorphism in their arithmetical operators, meaning that one can say $(3 + 4)$ or $(3.5 + 4.1)$ in order to add two integers or two floating-point numbers. The problem with languages that place limits on 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 some languages (e.g., Lisp and Smalltalk), all functions can be overloaded, so that one can create many types of objects that can be used interchangeably (e.g., many different objects can handle the `play` message in their own ways).

In O-O languages, the receiver of a message (i.e., the object to which the message is sent) determines what method to use (how) to respond to a message. In this way, all the various types of (e.g.) musical events and event collections can all receive the message `play` and will respond accordingly by performing themselves, although they may have very different methods for doing this. We have already seen examples of this in the ways that SC unit generator objects all understand the message `value`.

Example: Siren EventGenerators

The event generator package provides 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, ostinati, or compositional algorithms. Each event generator subclass knows how it is described—e.g., a chord with a root and an inversion, a trill with two notes that it repeats, or an osti-

nato with an event list and repeat rate—and can perform itself once or repeatedly, acting like a function, a control structure, or a process, as appropriate.

Some event generators describe relationships among events in composite event lists (e.g., chords described in terms of a root and an inversion), while others describe melismatic embellishments of—or processes on—a note or collection of notes (e.g., mordents). Still others are descriptions of event lists in terms of their parameters (e.g., ostinati). Most standard examples (chords, ostinati, rolls, etc.) above can be implemented in a simple set of event generator classes; the challenge is to make an easily-extensible framework for composers whose compositional process will consist of enriching the event generator hierarchy.

All event generators can either return an event list, or they can behave like processes, and be told to play or to stop playing. We view this dichotomy—between views of event generators as functions versus event generators as processes—as a part of the domain, and differentiate on the basis of the musical abstractions. It might, for example, be appropriate to view an ostinato as a process, or to ask it to play thrice.

Shared behavior of EventGenerators

Every event generator class has behaviors for the creation of instances, including, normally, some terse description formats (see the examples below). All event generator instances must provide a way of returning their events—the `eventList` method. Process-like event generators such as ostinati also provide start/stop methods for their repeated performance. The examples below illustrate other event generator behaviors.

SC EventGenerators

The example code in SC implements a simplified version of the Siren System’s event generator hierarchy (Pope 1989), with roll, trill, and several kind of stochastic “cloud” classes. All of the examples use a simple FM instrument for synthesis.

The class `EventGenerator` is abstract, meaning that it does not have enough knowledge to be used directly (instances of it would be useless), but that it serves as the superclass of several other classes. It has instance variables for duration, amplitude, pitch, and repetition rate, and methods for accessing these variables (the `get` and `set` methods), as well as several useful processing methods.

A roll object is an event generator that can play a simple repeating note, as in a drum roll. The GUI lets you choose the total duration of the roll, the rate of repetition, the amplitude, and the pitch. A trill is a roll that alternates between two pitches, which can be set from the lower values of the two pitch sliders in the GUI. (Actually, trills can have

more than two notes and will cycle through them, but that function is not supported by the simple demonstration GUI.)

Clouds are stochastic (i.e., semi-random) event generators where you can set a pitch range (in the case of a cloud object) or pitch set (in the selection cloud object) to select from. A dynamic cloud is simply a cloud with starting and ending pitch ranges between which it interpolates. If you set the ranges to be the same, it's the same as a cloud. If you set both ranges to single values, you get a scale. Try experimenting with different ranges (e.g., wide initial range and single-pitch final range).

Selection clouds use pitch sets rather than ranges; this means, for example, that you can select notes from a specific chord or mode. A dynamic selection cloud is a selection cloud that interpolates between selecting the pitches in its initial and final pitch sets. The class hierarchy of the basic event generators are shown in the figure.

```

EventGenerator
  Roll
    Trill
      Cloud
        DynamicCloud
          SelectionCloud
            DynamicSelectionCloud
  
```

Figure 30. Basic EventGenerator class hierarchy

The GUI for this program lets the user select between 6 basic kinds of event generator, and to set several parameters for the chosen class. The top three sliders are easy to understand; they set the event generator's total duration, its event rate (notes per second), and its amplitude.

The two pitch sliders are interpreted according to which kind of EventGenerator you choose. They are both range sliders, meaning that a numerical interval can be selected by dragging the mouse from left to right over them. For a roll event generator, only the lower value of the *Pitch1* slider is used; for a trill, the lower values of the two pitch sliders determine the trill's pitches.

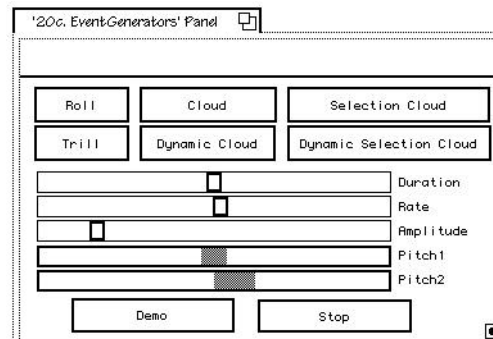


Figure 31. EventGenerator example GUI

For a cloud, the *Pitch1* slider determines the pitch range, and for a dynamic cloud, the *Pitch1* and *Pitch2* ranges are used as the starting and ending pitch ranges, respectively. The selection cloud object's demo function prompts you for a list of pitches (MIDI key numbers). The suggested range is 24 - 96. You can type 1 or more numbers into this list

(with no punctuation or list brackets, e.g., 36 38 40 41 43). For a dynamic selection cloud, you will be prompted for initial and final pitch sets. If you do not provide an answer, the program will use its own default sets.

The program below implements the complete (mini-) EventGenerators class hierarchy, and includes a number of demo functions. The instrument functions are not shown here, but they are included in the on-line source file. The program is broken up into several blocks, the first of which consists of the abstract class EventGenerator.

```
(*****
EventGenerators: An example of OOP in SC.
This is a simple translation of the basics of the "EventGenerators" package
from Smalltalk to SC. See reference (Pope 1989) for details.

This file includes several common event generators (egens for short) such as
roll, trill, and several types of stochastic "clouds." There are several test
functions after the class definitions that demonstrate the EGen's with parameters
taken from GUI sliders, and also a demo function that plays a small "composition"
described using egens.

Stephen Travis Pope -- stp@create.ucsb.edu -- 1997.02.24

*****)
```

```
defaudioout L R;

(*****
EventGenerator Abstract Class
This is the "root" of a family of classes that implement simple event structures.
Standard event parameters are [start, instr, dur, amp, pitch], though additional
(instrument-specific) parameters can be added using the addParameters message.
*****)
```

```
class EventGenerator {
  -- Instance variables (= arguments to the constructor function).
  --
  arg dur, amp, pitch, rate, events;
  --
  -- Dur is the total duration of the EGen.
  -- Rate is repetition rate for events within an EGen.
  -- The duration of the single events is 1 / rate.
  -- The number of events is dur * rate.
  -- Depending on the subclass you use, the pitch may be a single value, a list of [min max],
  -- a set (list) of legal values, or a 2-D list with starting and ending ranges or selection sets.
  -- Events is a list of lists of parameters.

  -- There is no initialization code in this class.
  -- (Otherwise, it would go here.)
```

```

-- Instance variable accessing methods.
--
method getPitch { ^pitch }
method getDur { ^dur }
method getAmp { ^amp }
method getRate { ^rate }
method setPitch { arg val; pitch = val }
method setDur { arg val; dur = val }
method setAmp { arg val; amp = val }
method setRate { arg val; rate = val }

method basicEvents { ^events } -- Low-level, private methods.
method setEvents { arg val; events = val }

-- Answer the receiver's events (with lazy [i.e., last-minute] initialization).
--
method getEvents {
    -- If the receiver has no events yet,
    if this.basicEvents == \nil then
        -- Post a message to the transcript,
        [\Generating this.type].post;
        -- and generate the events now.
        this.setEvents(this.generateEvents);
    end.if;
    ^events;
}

-- Generate the receiver's events (this will be overridden in subclasses).
--
method generateEvents {
    ^[] -- Answer an empty list in the abstract class.
        -- The subclasses will override this.
}

-- Add additional event parameters using a closure argument.
-- The closure must answer a list that will be concatenated to the
-- event's parameter list. (See the demo functions below for examples of using this.)
--
method addParameters {
    arg closure;

    newEvents = this.getEvents.collect({
        arg item, index;
        ^((item $ closure.value);
        ));
    this.setEvents(newEvents);
}

```

```

-- Play the receiver's events on the given voice at the given time
-- with additional parameters generated by the "args" closure (may be nil).
--
method play {
  arg voice, start, args;          -- voice is an instrument function reference;
                                   -- start is an offset time (a number).
                                   -- args is a closure that answers a list of
                                   -- additional parameters when evaluated.
  if args != \nil then             -- If there are additional parameters,
    this.addParameters(args);      -- add them.
  end.if;
  this.getEvents.forEach({        -- Loop through the events.
    arg item, index;              -- Each event is actually just a list of parameters.
                                   -- Add the start time to the event's start.
    item.put(0, ((item @ 0) + start));
    item.put(1, voice);           -- Assign the given voice to the event.
    -- item.post;                 -- Print event to message window (for debugging).
    item.sched;                   -- Schedule the event.
  })
}

-- Apply a function closure to the receiver egen's events.
--
method apply {
  arg function;                   -- A function to apply to the receiver's events.

  this.getEvents.forEach(function);
}

-- End of Class EventGenerator

```

Code Example 43. EventGenerator Class Definition

The EventGenerator class given above has all of the features that we need for our musical structure objects; it has the instance variables for its duration amplitude, etc., as well as methods to get and set (query and assign) these instance variables. What's missing is just a couple of key methods such as generateEvents. These will be refined in its subclasses, which will be introduced next.

```

(*****
  Roll Concrete Class (a subclass of EventGenerator)
  This represents a simple repeating note (e.g., a drum roll)
  *****)

```

```

class Roll : EventGenerator {
  -- The class does not add any new instance variables or initialization code.

```

```

-- Repeat a single note to get a roll.
-- We only have to override the generateEvents method; everything else is inherited!
--
method generateEvents {
  var notes, time, count, note_dur;

  time = 0;
  count = 0;
  notes = [];
  note_dur = 1/rate;
  count = dur * rate;

  for i = 0; i < count; i = i + 1; do
    notes.add([time, \nil, note_dur, amp, this.nextPitch(i)]);
    time = time + note_dur
  end.for;
  ^notes

-- Answer the pitch to use for the receiver roll.
--
method nextPitch {
  arg counter;

  ^pitch;
}

(*****
  Trill Concrete Class (a subclass of Roll)
  This represents a pair of alternating notes.
  The pitch instance variable is assumed to be a list of two or more pitch values.
  *****)

class Trill : Roll {

  -- Here, we only need to override the nextPitch method.
  --
  method nextPitch {
    arg counter;

    ^pitch @@ counter;
  }
}

```

Code Example 44. Roll and Trill Subclasses

Roll and Trill are two very simple EventGenerator subclasses, but they serve to demonstrate how you write concrete EventGenerators, and also to demonstrate the power of OOP. Because it inherits from EventGenerator, Roll only has to define a generateEvents method. It does so using a method called nextPitch to determine what pitch to play. While this method is rather useless in a roll (i.e., it always answers the same pitch), using it in the generateEvents method allows us to define the Trill as being a Roll that refines the nextPitch method only (and inherits all the rest of its behavior from Roll and EventGenerator). Note that the Trill is not limited to two pitches only, the pitch list could just as well have more than two elements in it.

The next set of EventGenerator classes I'll show you are the "clouds" that use random selection from a pitch interval or from a given pitch set.

```
(*****
  Cloud Concrete Class
  This represents a "cloud" of events in a given pitch range (pitch1-to-pitch2).
  *****)

class Cloud : EventGenerator {
  -- Cloud adds two new instance variables beyond those defined in EventGenerator.
  arg base, range;          -- lowest pitch and pitch interval.

  -- Initialization code
  --
  base = pitch @ 0;          -- pitch is a list of [min max].
  range = (pitch @ 1) - base;

  -- Select notes from the given range.
  --
  method generateEvents {
    var notes, time, count, note_dur, factor;

    time = 0;
    count = 0;
    notes = [];              -- Empty list of notes.
    note_dur = 1/rate;
    count = dur * rate;      -- How many notes to make.

    count.timesRepeat({      -- Loop to add notes.
      factor = time / dur;    -- The completion factor (goes from 0 to 1
                              -- over duration of the EGen).
      notes.add([time, \nil, note_dur, amp, this.nextPitch(factor)]);
      time = time + note_dur
    });
    ^notes
  }
}
```

```

-- next_pitch method (It takes a completion factor in these classes).
--
method nextPitch {
  arg factor;                                -- factor is ignored here.
  ^(base + range.rand);                      -- Answer a random pitch in the chosen range.
}
-- End of Cloud.

(*****
  SelectionCloud Concrete Class
  This represents a "cloud" of events with pitches chosen at random from a given pitch set
  ("pitch" is now a list of allowed pitches).
  *****)

class SelectionCloud : Cloud {

  -- next_pitch method
  -- Select from the given pitch set
  method nextPitch {
    arg factor;                                -- factor is ignored here.
    ^pitch.choose;                             -- Pitch is assumed to be a list of possible pitches
  }
-- End of SelectionCloud.

(*****
  DynamicCloud Concrete Class
  This represents a "cloud" of events in a pitch range that changes over time
  (pitch is a list of the starting and ending pitch ranges [[min1 max1][min2 max2])).
  *****)

class DynamicCloud : Cloud {
  -- This adds four more instance variables.
  arg min1, max1, min_diff, max_diff;

  -- Initialization code
  -- Pitch is a 2-D list ([[min1 max1] [min2 max2]])
  --
  min1 = (pitch @ 0) @ 0;                      -- Get the starting pitch range.
  max1 = (pitch @ 0) @ 1;
  min_diff = ((pitch @ 1) @ 0) - min1; -- Calculate the difference between the starting
  max_diff = ((pitch @ 1) @ 1) - max1; -- and ending min/max pitches

  -- nextPitch method answers a random value in the moving pitch range.
  --
  method nextPitch {
    arg factor;

```



```

    var min, max;

    min = min1 + (min_diff * factor); -- Use the completion factor to move through
    max = max1 + (max_diff * factor);-- the pitch range.
    ^min + (max - min).rand;
  }
} -- End of DynamicCloud

(*****
  DynamicSelectionCloud Concrete Class
  This represents a "cloud" of events with pitches selected from the given starting
  and ending pitch sets (pitch = [[a b c][x y z]])
  *****)

class DynamicSelectionCloud : DynamicCloud {
    -- Randomly-chosen pitches between starting/ending pitch sets

    method nextPitch {
        arg factor;
        var random;

        random = 1.0.rand;          -- Compare the completion factor to a random.
        if random > factor then      -- Choose from the initial or final set based on that.
            ^(pitch @ 0).choose;
        else
            ^(pitch @ 1).choose;
        end.if;
    }
}

```

Code Example 45. Cloud Classes

The Cloud classes are similar to the Rolls in that they all share (reuse) the same method for generateEvents, and only refine (override) the nextPitch method with their own versions that implement their specific behavior. The next set of functions are the demonstration methods; these are sent by the demo buttons on the EGenS GUI screen (shown above).

```

(*****
  Test functions sent by GUI buttons -- Read GUI item values and create and play EGenS
  *****)

-- Answer a list of [dur rate amp] read off of the GUI sliders.
--
getGUIValues {
    var list;

```

```

    list = [];
    list add(8.getItemValue);    -- Read the duration slider.
    list add(9.getItemValue);    -- Read the rate slider.
    list add(10.getItemValue);   -- Read the amplitude slider.
    ^list
}

-- Play a simple demo of a roll egen.
-- Use the base (lower) value of the pitch 1 slider as the pitch.
--
playRoll {
    var dur, amp, pitch, rate, roll;
                                -- Read GUI sliders and use multiple assignment.
    #dur rate amp = getGUIValues;
    pitch = 11.getItemValue.    -- The pitch is the base value of slider 11.

    roll = Roll(dur, amp, pitch, rate); -- Create a Roll object.
                                -- Play it passing along a closure that answers a
                                -- list of parameters for the FM instrument
    roll.play(`fm_instr, 0,      -- (See the play method in EventGenerator.)
              -- index ratio pos att dec i_att i_dec
              { ^[6, 1, (1.0.bilin), 0.004, 0.046, 0.001, 0.03] } );
}

-- Play a trill example.
-- The pitches are the base values of the two pitch sliders.
--
playTrill {
    var dur, amp, pitch1, pitch2, rate;
                                -- Read GUI sliders and use multiple assignment.
    #dur rate amp = getGUIValues;
    pitch1 = 11.getItemValue.    -- The pitches are the bases of sliders 11 and 12.
    pitch2 = 12.getItemValue;
                                -- Create and play a Trill object.
    Trill(dur, amp, [pitch1 pitch2], rate).play(`fm_instr, 0,
          { ^[4, 1, (1.0.rand2), 0.008, 0.04, 0.001, 0.03] });
}

-- Play an example on a random cloud egen.
-- The pitch range is taken from the upper and lower bounds of the pitch 1 slider.
--
playCloud {
    var dur, amp, pitchmin, pitchmax, rate;
                                -- Read GUI sliders and use multiple assignment.
    #dur rate amp = getGUIValues;
    pitchmin = 11.getItemValue;  -- get the lower value of the range slider.
    pitchmax = 11.getItemValue2; -- get the upper value of the range slider.

```

```

-- Create and play a Cloud object.
Cloud(dur amp, [pitchmin pitchmax], rate).play(`fm_instr, 0,
  {[2, 1.04, (1.0.bilin), 0.006, 0.044, 0.001, 0.03]});
}

-- Play a selection cloud using a list of pitches queried from the user.
--
playSCloud {
  var dur, amp, pitches, rate;
  #dur rate amp = getGUIValues; -- Read GUI sliders and use multiple assignment.

  -- Prompt the user for a list of pitches to select among
  -- (MIDI key values between 24 and 96 suggested).
  pitches = getListFromUser("Please type in a list of pitches to select from.");

  if pitches.size = 0 then -- If the user didn't type in any pitches,
    pitches = [52 53 54 55 56 57] -- use these defaults.
  end.if;

  -- Create and play a SelectionCloud object.
  SelectionCloud(dur, amp, pitches, rate).play(`fm_instr, 0,
    {[2, 1.04, (1.0.bilin), 0.006, 0.044, 0.001, 0.03]});
}

-- Play a dynamic cloud based on the intervals set on the two pitch range sliders.
--
playDCloud {
  var dur, amp, pitchmin1, pitchmax1, pitchmin2, pitchmax2, rate;

  #dur rate amp = getGUIValues; -- Read GUI sliders and use multiple assignment.

  pitchmin1 = 11.getItemValue; -- Get the initial and final pitch intervals.
  pitchmax1 = 11.getItemValue2;

  pitchmin2 = 12.getItemValue;
  pitchmax2 = 12.getItemValue2;

  -- Create and play a DynamicCloud object.
  DynamicCloud(dur, amp, [[pitchmin1 pitchmax1][pitchmin2 pitchmax2]], rate)
    .play(`fm_instr, 0,
      {[2, 1.04, (1.0.bilin), 0.006, 0.044, 0.001, 0.03]});
}

-- Demonstrate the dynamic selection cloud by asking the user for two lists of pitches.
--

```

```

playDSCloud {
  var dur, amp, pitches1, pitches2, rate;
                                -- Read GUI sliders and use multiple assignment.
  #dur rate amp = getGUIValues;
                                -- Prompt the user for 2 lists of pitches, supplying
                                -- defaults if he/she answers with the empty list.
  pitches1 = getListFromUser("Please type the initial list of pitches.");
  if (pitches1.size = 0) then
    pitches1 = [48 50 52 54 56]
  end.if;

  pitches2 = getListFromUser("Please type the final list of pitches.");
  if (pitches2.size = 0) then
    pitches2 = [72 73 73]
  end.if;

                                -- Create and play a DynamicSelectionCloud object.
  DynamicSelectionCloud(dur, amp, [pitches1 pitches2], rate)
    .play(`fm_instr, 0,
      {[2, 1.04, (1.0.bilin), 0.006, 0.044, 0.001, 0.03]});
}

```

Code Example 46. EventGenerator Test Methods

Finally, below is a simple “composition” described wholly in terms of EventGenerators. It is intended to show the terseness and flexibility of these objects when used as a music notation, rather than being a compelling musical experience. The example consists of a background ostinato pattern of two alternating trills over which several other EventGenerators are layered. It is triggered by the **Demo** button on the GUI.

```

(*****
  Play a small EGen "composition"
  *****)

playDemo {
  var egen, egenDur;
                                -- Play a sequence of slow trills in the background.
  for i = 0; i < 10; i = i + 1; do
    Trill(1.75, 0.1, [48 50], 4).play(`fm_instr, (i * 3.5),
      {[1, 2, (1.0.bilin), 0.008, 0.04, 0.001, 0.03]});
    Trill(1.75, 0.1, [49 51], 4).play(`fm_instr, ((i * 3.5) + 1.75),
      {[1, 2, (0.6.bilin), 0.008, 0.04, 0.001, 0.03]});
  end.for;

                                -- Make a 15-sec. roll that has a crescendo and slow pan.
  egenDur = 15;

  egen = Roll(egenDur, 1, 36, 14);-- Create the roll object.

```

```

        -- Make a crescendo function that will scale event amplitude values
        -- by a factor that goes from 0 to 1 over the duration of the egen.
egen.apply({ arg evt;
    evt.put(3, ((evt @ 3) * (((evt @ 0) / egenDur) ** 6))));

        -- Add parameters specific to the FM instrument.
egen.addParameters({^[6, 1.02, 0, 0.008, 0.04, 0.001, 0.03]});

        -- Now make a R/L pan over the dur of the roll
        -- (position is parameter 7).
egen.apply({arg evt;
    evt.put(7, (1 - (((evt @ 0) / egenDur) * 2))));

        -- Play the roll with no additional parameters
egen.play(`fm_instr, 0, \nil);

        -- End with a few short clouds at the end.
SelectionCloud(1, 0.1, [72 74 76], 12).play(`fm_instr, 16,
    {^[1, 1, (1 - 0.3.rand), 0.008, 0.04, 0.001, 0.03]});

SelectionCloud(1, 0.1, [72 74 76], 12).play(`fm_instr, 18,
    {^[1, 1, (1 - 0.3.rand), 0.008, 0.04, 0.001, 0.03]});

SelectionCloud(1, 0.15, [78 80 82], 14).play(`fm_instr, 22,
    {^[3, 1, (0.3.rand - 1), 0.008, 0.04, 0.001, 0.03]});

SelectionCloud(1, 0.15, [78 80 82], 15).play(`fm_instr, 24,
    {^[4, 1, (0.3.rand - 1), 0.008, 0.04, 0.001, 0.03]});

DynamicSelectionCloud(2.5, 0.1, [[78 80 82][80 82 83 84 86]], 10)
    .play(`fm_instr, 29,
    {^[1, 1.1, 0.3.rand, 0.014, 0.04, 0.006, 0.03]});

SelectionCloud(1, 0.1, [83 84 86], 11).play(`fm_instr, 34,
    {^[1, 1.1, 0.3.rand, 0.014, 0.04, 0.006, 0.03]});

        -- Stop the scheduler at the end.
36 `dspKill].sched;
\Done.post
}

```

Code Example 47. A Small ‘Composition’ Using EGens

More kinds of EventGenerators can be constructed within this framework, and the abstract class could also be extended, for example with a more sophisticated notion of parameter addition and function application.

There are innumerable other applications of object-oriented programming within SC; these preliminary examples should serve simply to whet the reader’s appetite.

6.4. Algorithmic Composition

One of the most unique features of SC is the blur between composition and synthesis functions. Any SC function you write can generate output samples, read/write score or data files, or access shared data objects. In this Part, we will introduce several techniques for music modeling and algorithmic composition. This topic is certainly worthy of an entire book of its own.

Passing Data between Collaborating Functions

To introduce the technique of separating compositional and synthesis functions into collaborating *co-routines*, we first present James McCartney’s “harmonics” example. There are two functions running in parallel in this example; one—the “composer”—periodically resets the fundamental frequency used by the other—the “performer.” The two functions communicate via the shared global variable `fn`; it represents the fundamental frequency chosen by the composer function. The composer recalculates the fundamental every 5 seconds. The performer function plays random overtones of this frequency at the rate of 7 notes per second.

```
-- Make random harmonic overtones of a periodically changing fundamental
-- (James McCartney's "harmonics" example.)
--
defaudioout L, R;
deftable wave, env;                                -- Declare a wave table and envelope function.

var fn = 50.0;                                       -- Global shared fundamental frequency.
                                                    -- Starting value = 50 Hz.

start {
    fundamental;
    overtones;
}                                                    -- Run both functions in parallel from start.
                                                    -- Start the composer function.
                                                    -- Start the performer function.

-- "Composer" function -- Reset and print the fundamental "fn" every 5 seconds.
--
fundamental {
    fn = 50.0 + 20.0.rand2;                          -- Compute a new value for fn (30 - 70 Hz).
    fn.post;                                           -- Print it to the message transcript.
    [5.0 thisFunc].sched;                             -- Repeat function in 5 seconds.
}

-- "Performer" function -- Play random overtones of frequency fn.
--
overtones {
    var cosc env chan;                                -- Declare local variables.
```

```

-- Osc with freq. (x * fn) (i.e., overtones of fn).
-- (Remember that 16.rand answers an integer.)
cosc = Acoscil(wave, fn * (16.rand + 1), 0.1);
-- Envelope generator (notes last 1.4 seconds).
env = Ktransient(env, 1.4, 0.2, 0, `dspRemove);
chan = [L R].choose; -- Random channel selection.

-- DSP loop.
{ (cosc. * env.).out(chan) }.dspAdd;

[0.15 thisFunc].sched; -- Repeat in 0.15 seconds.
}

```

Code Example 48. Random Overtones of a Computed Fundamental

This example demonstrates the structure of separating out composition functions that generate musical “base data” and store it in some shared variables, and synthesis functions that read this base data and generate their output using it. In this case, the two are independently running routines that communicate via global data.

Using Lists to Create Patterns

A Simple Pattern Repeater

The next example plays rhythmic patterns of repeated notes (e.g., 3 eighth notes on f, then 5 quarter notes on a, then 2 half notes on e-flat, etc.). The start function defines three data lists that hold the sequences of pattern lengths (the number of times a note is repeated), durations (in virtual beats or “clock ticks”), and pitches (a pentatonic scale in this example). In the first version, the pattern length and duration list series are of the same length (to make it easier to hear what’s going on). The player function cycles repeatedly through the length table (using @@ for wrap-around indexing), playing notes using the durations from the duration table inside a loop that is repeated based on the length data list. It chooses at random from the list of pitches. The effect is an ever-changing pattern of repeats. There are two instruments used (named *fm_instr* and *wave_instr*), between which we choose at random (they are not printed here).

```

-- Repeat rhythmic pulse phrases on two instruments.
--
defaudioout L, R;
--
-- Globals for shared data.
var i, d, lengths, durations, frequencies, tempo;

start {
  -- Set up tables of repeats, durs, and freqs.
  lengths = [3 5 2 4]; -- Number of notes repeated per group.
  durations = [6 3 8 5]; -- Note durations in “ticks.”
  frequencies = [48 50 53 55 57 60 62]; -- Pitches are pentatonic key numbers.
}

```

```

    i = 0;
    d = 0;
    tempo = 0.03;
    player;
}

-- Player function repeats notes in groups.
--
player {
    var count, dur, freq, j, instr;

    count = lengths @@ i;
    dur = (durations @@ d) * tempo;
    freq = frequencies.choose;
    instr = [fm_instr `wave_instr].choose;

    for j = 0; j < count; j = j + 1; do
        [now + (j * dur), instr, dur, freq, 2, 1].sched;
    end.for;

    i = i + 1;
    d = d + 1;
    [(count - 1) * dur, thisFunc].sched;
}

... instruments not included here ...

```

Code Example 49. Simple Pattern Repeater

Performing this basic example makes the pattern player process clear; for a more interesting structure, try using lists of lengths and durations with different sizes, as in,

```

lengths = [3 5 2 6 4];
durations = [6 4 8 5 3 5];

```

One could also change the player function to step through the pitch list in order and select at random from the duration list, for example, or write a co-routine that periodically changes one or more of the lists. Other potential extensions include using some of the I/O and GUI functions we presented above to make this instrument read its lists from text files, or to allow the user to edit (or play) the lists in real time performance.

Using Pattern Streams

Pattern streams are one of the most powerful abstractions for algorithmic composition in SC. These are similar to the objects used for higher-level musical structures in related

music languages such as HMSL, CommonMusic, DMix or the MODE. Pattern streams allow one to define complex deterministic or stochastic structures and to control their repetition and evolution.

The simplest pattern stream is Sseq (sequential pattern); it consists of a data list that it reads through in sequence, with a function that can alter the list's contents when the pattern repeats (i.e., when you get to the end of the list). As an introductory example, suppose we want a function that generates data like

1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9 ...

To create this, we would define a sequential pattern stream that “shifts” itself one step higher with each repetition. The following example illustrates this.

A “Staircase” Pattern Stream

To create a pattern stream, we give it an initial list of data, a number of repetitions to perform, and some (optional) functions to perform on start-up, on repeat, or on completion. This example defines a function that removes the first item of the list and adds a new one to the end on each repetition.

```
-- Pattern Stream Composition Example -- Staircase noise pattern
--
defaudioout L, R;
defenvelope env_func;      -- Envelope function.

-- Define a function to remove the first item of a list and add a new item (that's
-- 2 greater than the last item) to the end of the list.
--
list_step {
  arg list, count;          -- The list and repetition count are the default arguments.

  var length;               -- The length of the list.

  if (list @ 0) > 100 then -- Stop when the lower limit is 100.
    dspKill;
  end.if;
  list.removeAt(0);         -- Remove the first (zero-th) element.
  length = list.size;       -- Get the length of the list.
  list.add((list @ (length - 1)) + 2); -- Add a new item at the end that's 2 greater
                                -- than the current last item.
  list.post;               -- Print the list.
  ^list                    -- Answer the new list.
}

var pattern, lfo;           -- Declare the global pattern stream and LFO.
```

```

-- Start function sets up pattern stream and calls player function.
--
start {
    -- Create a whole-tone pattern stream for the pitches
    pattern = Sseq([36 38 40 42 44 46 48],
        100, -- Up to 100 repeats.
        \nil, -- No start function
        `list_step -- Repeat function = list_step (as a function reference)
        \nil); -- No stop function

    lfo = Psinosc(0.33, 0); -- define polled LFO at 1/3 Hz for position.

    player; -- start player function
}

-- Play the filtered noise instrument with the ever-increasing pitch pattern.
--
player {
    var dur, bw, freq;

    dur = 0.12; -- Duration of notes (1/8 seconds).
    bw=15; -- Bandwidth of noise filter.
    freq = pattern.value.midicps; -- Get the frequency from the pattern stream.
    -- The pattern stream will automatically call its repeat
    -- function and reset its contents at the end of the list
    -- (every 7 notes).
    -- Play a note, get position from LFO.
    --      dur ampl  freq BW  position
    noise_instr(dur, 1.0,  freq, bw, lfo.value);

    [dur, thisFunc].sched; -- Repeat notes 8 / second.
}

-- Noise instrument as above...

```

Code Example 50. Pattern Stream Example

Random Pattern Streams and Update Functions

There are many other kinds of pattern streams, and interesting behaviors that can be achieved with streams on multidimensional patterns, or pattern streams with more interesting repeat functions. The final program in this chapter shows the use of a semi-random pattern in consort with the above example.

In this case, we use the `Srand` pattern stream constructor and give it no start/repeat/stop functions to control its behavior. Instead, when we get its value, we compare that value with the special symbol `\end`. This is the default return value from a pattern stream if it runs out of data and does not have a pre-defined repeat function. The player

function now steps the list “by hand” and creates a new Srand pattern stream for each iteration of the random list.

```
-- Pattern Stream Composition Example 2.
-- 2 pattern streams, one sequential and one random-order.
--
defaudioout L, R;

var pattern1, pattern2, lfo, ;ist;      -- Declare global patterns, lfo, and global data list.

-- Start function sets up patterns and calls player function.
--
start {
    list = [34 36 38 40 42 44 46 48];-- Create the whole-tone data list.
                                   -- Pattern1 is sequential as above.
    pattern1 = Sseq (list, 100, \nil, `list_step, \nil);
    pattern2 = Srand(list, 8);      -- Pattern2 is random-order; it ends after 8 values.
                                   -- (It has no repeat function.)
    lfo = Psinosc(0.33, 0);        -- Define LFO for position.
    player;                        -- Start player function (it repeats).
}

-- This player function plays two streams of events.
--
player {
    var dur, amp, bw, freq;        -- Declare local variables.

    dur = 0.12;                   -- Duration of notes.
    bw=10;                        -- Noise filter bandwidth.
    amp = 0.5;                    -- Amplitude.
    freq = pattern1.value.midicps; -- Get the value of pattern1.
    noise_instr(dur, amp, freq, bw, lfo.value); -- Play the first note.

    freq = pattern2.value;        -- Get the 2nd frequency using pattern2.
    if freq == \end then          -- If at the end of the pattern, reset and restart.
        list = list_step(list);
        pattern2 = Srand(list, 8);
        freq = pattern2.value.midicps;
    end.if;

    noise_instr(dur, amp, freq.midicps, bw, lfo.value.neg);-- (because of lfo.value.neg).
    [dur, thisFunc].sched;        -- repeat player
}

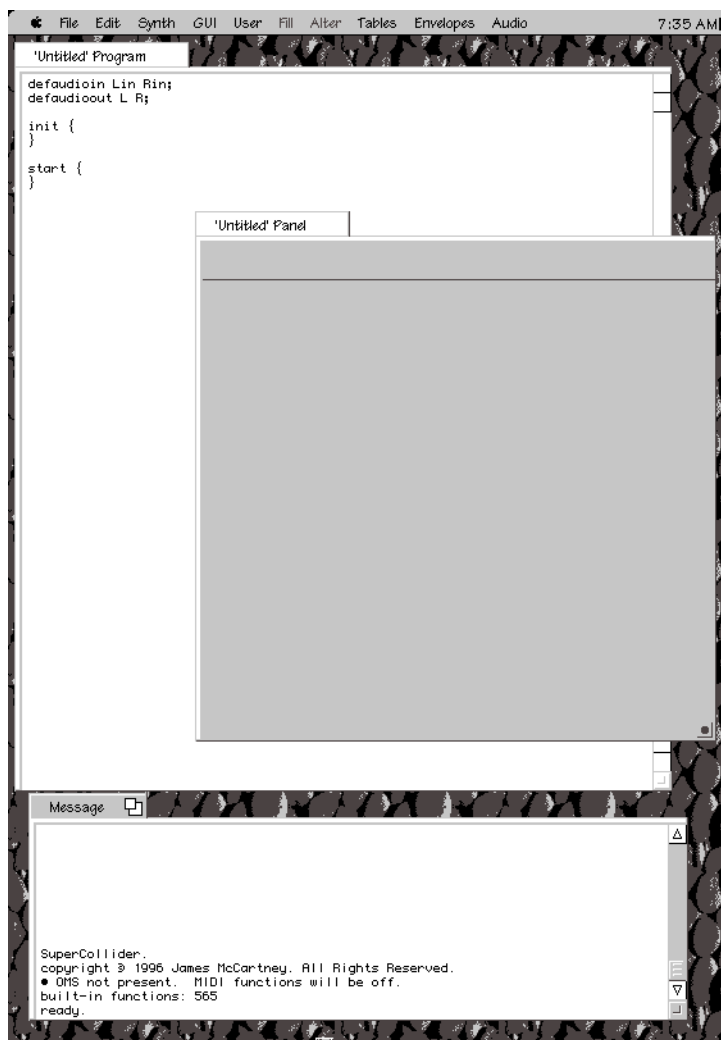
... noise instrument as above ...
```

Code Example 51. Dual Pattern Stream Example

There are other types of pattern streams, and also a facility called pattern threads, which are like pattern streams, but that manipulate the execution order of functions rather than lists of data. These are quite useful for building your own schedulers.

Another advanced exercise would be to create a program that uses pattern streams of EventGenerators, for example Trills.

In this section, we surveyed several intermediate SC programming techniques. Beyond this introduction, the reader is encouraged to experiment with as many of the different areas covered as possible. Advanced users are equally comfortable using SC's file I/O, GUI interfacing, interactive performance, and other programming techniques.



Part 7. Conclusions

7.1. Summary

In this course book, I have introduced the topic of software sound synthesis, and presented the SuperCollider language and programming system. The examples demonstrated a number of sound synthesis and processing techniques, and illustrated several advanced SuperCollider programming techniques for production, processing, and algorithmic composition.

Not Covered Here

There are a number of other interesting features of the SC system that are deemed to be outside the scope of this basic tutorial. These include tempo bases and functions, graphical drawing functions, pattern threads, user-defined menus, and advanced GUI construction. Readers interested in these topics are referred to the SC manual and the on-line SC examples that come with the program distribution (or potentially to future versions of this book).

Call for Feedback

Please let me know what else you'd like to have here. Readers (and fellow SC developers) who submit working and well-documented code examples that are included in a future version will, of course, be acknowledged in the book.

7.2. How to Get SuperCollider and More Information

SuperCollider is a commercial software package. A free demonstration version (that has all of the relevant functions except the ability to save files) is available from the Internet FTP site,

`ftp://mirror.apple.com/mirrors/Info-Mac.Archive/gst/snd/super-collider-demo.hqx`

You can, in fact, work through the entire course presented here with the free version, but I strongly encourage all readers to support the further development of this excellent tool by buying a full version. To do this, send US\$ 250.00 (+ US\$ 10.00 for shipping) to,

James McCartney
3403 Dalton St.
Austin, TX 78745 USA

James's two active electronic mail addresses are `james@clyde.as.utexas.edu` and `james@lcsaudio.com`.

For More Information

There is a very useful electronic mail mailing list for SC users. Its address is,

`sc-users@lists.realtime.net`

To join this, send a mail message to `majordomo@lists.realtime.net` with the *contents*,

`subscribe sc-users`

(the subject of the letter is ignored).

An Internet FTP archive of user examples (including all of the examples presented in this book), can be found at the world-wide web URL,

`ftp://ftp.realtime.net/vendors/sc-users`

Appendix E. References

- Beauchamp, J. 1989. "The Computer Music Project at the University of Illinois at Urbana-Champaign: 1989." *Proceedings of the 1989 International Computer Music Conference*. San Francisco: International Computer Music Association. pp. 21-24
- Deitel, H. M., and P. J. Deitel. 1992. *C: How to Program*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Lansky, P. 1990. *CMix Release Notes and Manuals*. Department of Music, Princeton University.
- Lent, K, R. Pinkston, and P. Silsbee. 1989. "Accelerando: A Real-Time, General-Purpose Computer Music System." *Computer Music Journal* 13(4): 54-64.
- Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* 15(3): 41-49.
- Mathews, M. V. 1960. "Computer Program to Generate Acoustic Signals." Abstract in *Journal of the Acoustical Society of America* 32: 1493
- Mathews, M. V. 1961. "An Acoustical Compiler for Music and Psychological Stimuli." *Bell System Technical Journal* 40:677-694.
- Mathews, M. V. 1963. "The Digital Computer as a Musical Instrument." *Science* 142: 553-557.
- Mathews, M. V. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: MIT Press.
- McCartney, J. 1996. *SuperCollider A Real-Time Sound Synthesis Programming Language*. Program Reference Manual. Austin, Texas.
- Moore, F. R. 1990. *Elements of Computer Music*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Moore, F. R. and D. G. Loy. 1983. *CARL Release Notes, CARL Start-up Kit*. Computer Audio Research Lab, Center for Research in Computing and the Arts. La Jolla: University of California in San Diego.
- Pierce, J. R., M. V. Mathews and J.-C. Risset. 1965. "Further Experiments on the Use of the Computer in Connection with Music." *Gravesaner Blaetter* 27/8: 92-97.
- Pope, S. T. 1989. "Modeling Musical Structures as EventGenerators." *Proceedings of the 1989 International Computer Music Conference*. San

-
- Francisco: International Computer Music Association.
- Pope, S. T. 1992. "The Interim DynaPiano: An Integrated Tool and Instrument for Composers." *Computer Music Journal* 16(3): 73-91.
- Pope, S. T. 1993. "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal* 17(2): 23-54.
- Pope, S. T. 1996. "Object-Oriented Music Representation." *Organised Sound* 1(1): 55-68. Extended and revised in Roads, C, S. T. Pope, G. De Poli, and A. Piccialli, eds. 1997. *The Musical Signal*. Lisse, The Netherlands: Swets and Zeitlinger.
- Roads, C. 1996. *The Computer Music Tutorial*. MIT Press.
- Roads, C, S. T. Pope, G. De Poli, and A. Piccialli, eds. 1997. *The Musical Signal*. Lisse, The Netherlands: Swets and Zeitlinger.
- Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2): 23-38. reprinted in S. T. Pope, ed. *The Well-Tempered Object*. Cambridge, Massachusetts: MIT Press.
- Schottstaedt, W. 1992. *Common Lisp Music Documentation*. Reference manual available via Internet ftp from the clm directory on the host machine ccrma-ftp.stanford.edu.
- Tenney, J. C. 1963. "Sound Generation by Means of a Digital Computer." *Journal of Music Theory* 7: 25-70.
- Vercoe, B. 1996. *CSound Manual and Release Notes*. available via Internet ftp from the music directory on the host machine media-lab.mit.edu.

Appendix F. On-Line CodeExamples

This tutorial book is intended as the basis of an interactive on-line course in SuperCollider. All of the example programs presented above are available via the Internet for WWW or FTP file transfer from CREATE's Web site and FTP archive., The Web URL is <http://www.create.ucsb.edu/SC>.

The FTP address is

<ftp.create.ucsb.edu/pub/music/SC>.

The source files are stored in a Macintosh-format Stuffit archive with BinHex encoding. If you don't know what this means, or how to decode and unpack a BinHexed Stuffit file, consult a Macintosh initiate or a printed resource.

The source files have file names that start with their example numbers and include a description of what technique they demonstrate.

Files: (List to come)

00
01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Appendix G. SuperCollider Function List

(To come.)

Appendix H. Index

(To come.)