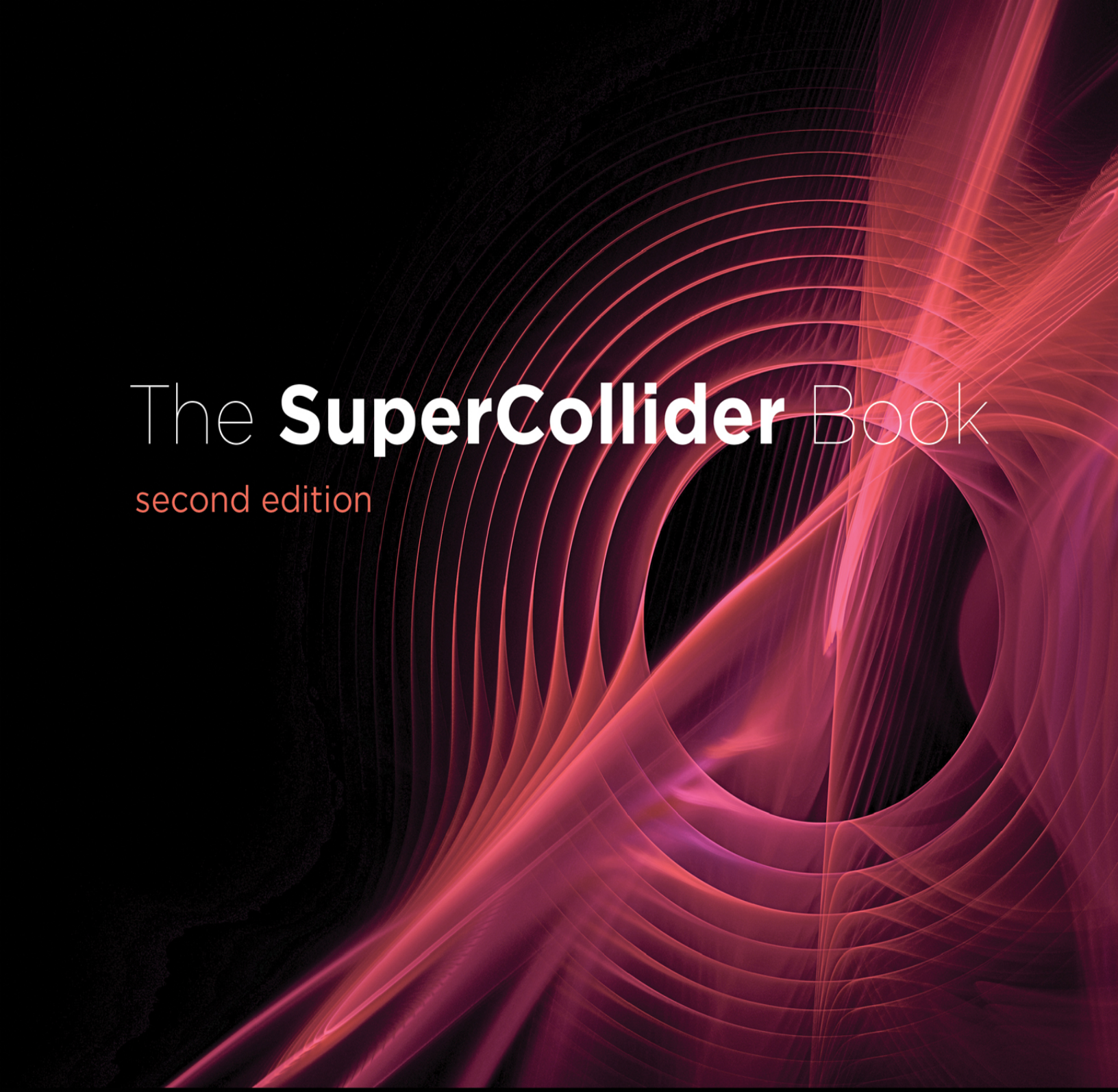


The SuperCollider Book

second edition

edited by Scott Wilson, David Cottle, and Nick Collins

foreword by James McCartney

The SuperCollider Book

second edition

edited by Scott Wilson, David Cottle, and Nick Collins

foreword by James McCartney

The MIT Press
Cambridge, Massachusetts
London, England

The MIT Press
Massachusetts Institute of Technology
77 Massachusetts Avenue, Cambridge, MA 02139
mitpress.mit.edu

© 2025 Massachusetts Institute of Technology

All rights reserved. No part of this book may be used to train artificial intelligence systems or reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

Library of Congress Cataloging-in-Publication Data

Names: Wilson, Scott, editor. | Cottle, David, editor. | Collins, Nick, editor. | McCartney, James, writer of foreword.
Title: The SuperCollider book / edited by Scott Wilson, David Cottle, and Nick Collins ; foreword by James McCartney.
Description: Second edition. | Cambridge, Massachusetts : The MIT Press, 2025. | Includes bibliographical references and index.
Identifiers: LCCN 2024022915 (print) | LCCN 2024022916 (ebook) | ISBN 9780262049702 (hardcover) | ISBN 9780262382717 (pdf) | ISBN 9780262382724 (epub)
Subjects: LCSH: SuperCollider (Computer file) | Software synthesizers. | Computer composition (Music) | Computer sound processing.
Classification: LCC ML74.4.S86 S86 2024 (print) | LCC ML74.4.S86 (ebook) | DDC 781.3/45133—dc23/eng/20240515
LC record available at https://lccn.loc.gov/2024022915
LC ebook record available at https://lccn.loc.gov/2024022916

EU product safety and compliance information contact is: mitp-eu-gpsr@mit.edu

d_r0

Contents

Foreword
James McCartney

Introduction to the Second Edition
Scott Wilson, David Cottle, and Nick Collins

PART I: TUTORIALS

1 Beginner's Tutorial
David Michael Cottle

2 The Unit Generator
Joshua Parmenter

3 Composition with SuperCollider
Scott Wilson and Julio d'Escriván

4 Ins and Outs: SuperCollider and External Devices
Marije A. J. Baalman, Miguel Negrão, Stefan Kersten, and Till Bovermann

PART II: ADVANCED TUTORIALS

5 Programming in SuperCollider
Iannis Zannos

6 Events and Patterns
Ron Kuivila

7 Just-in-Time Programming
Julian Rohrhuber and Alberto de Campo

8 Object Modeling
Alberto de Campo, Julian Rohrhuber, and Till Bovermann

PART III: EDITORS AND GUI

9 Installing, Setting Up, and Running the SuperCollider IDE
Norah Lorway

10 Alternative IDEs for SuperCollider
Konstantinos Vasilakos

11 SuperCollider on Small Computers

Matthew John Yee-King

12 The SuperCollider GUI Library
Eli Fieldsteel

PART IV: PRACTICAL APPLICATIONS

13 Sonification and Auditory Display in SuperCollider
Alberto de Campo, Julian Rohrhuber, Katharina Vogt, and Till Bovermann

14 Spatialization with SuperCollider
Marije A. J. Baalman and Scott Wilson

15 Machine Listening in SuperCollider
Nick Collins

16 Microsound
Alberto de Campo and Marcin Pietruszewski

17 Alternative Tunings with SuperCollider
Fabrice Mogini

18 Non-Real-Time Synthesis and Object-Oriented Composition
Brian Willkie and Joshua Parmenter

19 Stochastic and Deterministic Algorithms for Sound Synthesis and Composition
Sergio Luque and Daniel Mayer

PART V: PROJECTS AND PERSPECTIVES

20 Implementing New Language Syntax Using SuperCollider's Preprocessor
H. James Harkins

21 Interface Investigations
Thor Magnusson

22 SuperCollider in Japan
Takeko Akamatsu

23 Dialects, Constraints, and Systems within Systems
Julian Rohrhuber, Tom Hall, and Alberto de Campo

24 Artists' Statements
Juan Gabriel Alzate Romero, Helene Hedsund, Patrick Hartono, Norah Lorway, Andrea Valle, Marianne Teixidó, Neil Cosgrove, Shelly Knotts, Juan Sebastián Lach, Mileece, Sam Pluta, Ann Warde, and Anna Xambó Sedo

25 Machine Learning in SuperCollider
Chris Kiefer and Shelly Knotts

26 Notations and Score-Making
Tom Hall, Newton Armstrong, and Richard Hoadley

27 SCTweets: Character Matters
Fellipe M. Martins

PART VI: DEVELOPER TOPICS

28 The SuperCollider Language Implementation
Stefan Kersten

29 Writing Unit Generator Plug-ins
Dan Stowell and Christof Ressi

30 Inside scsynth
Ross Bencina

Subject Index
Code Index

List of Figures

Figure 1.1
Example of additive synthesis.

Figure 1.2
Nested commands for hypothetical lunch-producing code.

Figure 1.3
Futuristic (circa 1956) nested music.

Figure 1.4
VCO, VCF, VCA.

Figure 1.5
Synthesis example with variables and statements.

Figure 1.6
Phase modulation with modulator as ratio.

Figure 1.7
Synth definition.

Figure 1.8
Playback buffers.

Figure 1.9
Connecting controls with a bus.

Figure 1.10
Buffer modulation.

Figure 1.11
Random MIDI walk.

Figure 1.12
Random Crotale Walk.

Figure 1.13
Nested `do` to generate a 12-tone matrix.

Figure 1.14
Additive synthesis.

Figure 1.15
Example of additive synthesis.

Figure 1.16
Sonogram of additive synthesis example.

Figure 1.17
Physically modeled bells.

Figure 1.18
Generative sequences using arrays.

Figure 1.19
Offset and scale.

Figure 1.20
SinOsc offset and scaled for control.

Figure 1.21
PMOsc with sample and hold (latch).

Figure 1.22
It's just a bell.

Figure 2.1
Arrays and multichannel expansion.

Figure 2.2
Dynamics processing.

Figure 2.3
Triggering from within the server.

Figure 2.4
Triggers and sending values from the server back to the SuperCollider language.

Figure 2.5
Random-number generators and random seeding in the server.

Figure 2.6
GVerb.

Figure 2.7
Audio rate and control rate comparisons.

Figure 2.8
Signal routing optimizations.

Figure 2.9
Client-side calculations versus server-side calculations.

Figure 2.10
Interpolation methods in delays, and the processor and sonic differences.

Figure 3.1
A simple Routine illustrating a musical use of yield.

Figure 3.2
Using `Task` so you can pause the sequence.

Figure 3.3
Nesting tasks inside routines.

Figure 3.4
Using patterns within a `Task`.

Figure 3.5
Thanks to polymorphism, we can substitute objects that understand the same message.

Figure 3.6
Using “messaging style” with `Score`.

Figure 3.7
Executing one line at a time.

Figure 3.8
Play cues with a simple GUI.

Figure 3.9
A `SynthDef` for stereo buffer playback.

Figure 3.10
Recording the results of making sounds with SuperCollider.

Figure 3.11
A Server Window

Figure 3.12
A variable number of resonators with an automatically created GUI

Figure 3.13
Making a stuttering gesture using a geometric pattern.

Figure 4.1
Example of using the `Mouse` UGens.

Figure 4.2
Example of capturing keyboard events using the GUI.

Figure 4.3
Example of sending out MIDI using a `Task`.

Figure 4.4
Reading data from the SerialPort with a Routine.

Figure 4.5
The Modality GUI representation of the Korg nanoKONTROL2 MIDI controller.

Figure 4.6
Defining a very simple instrument with the Modality Toolkit.

Figure 4.7
Using named elements to decouple the instrument from the controller.

Figure 5.1
Some objects.

Figure 5.2
Receiver, message, method, return value.

Figure 5.3
Keyword arguments.

Figure 5.4
Grouping and precedence.

Figure 5.5
Statements.

Figure 5.6
Variable as a label pointing to a container.

Figure 5.7
Here, `nil` stands for the contents of an empty variable.

Figure 5.8
Variables can store objects that need to be used many times.

Figure 5.9
Variables can point to different objects during a process.

Figure 5.10
Three instances of `Point` with their instance variables.

Figure 5.11
`currentEnvironment`.

Figure 5.12
`topEnvironment` versus `currentEnvironment`.

Figure 5.13
Variables store only values, not other variables.

Figure 5.14
Assigning a variable to another variable stores its contents only.

Figure 5.15
Multiple use of a function stored in a variable.

Figure 5.16
Compiling and evaluating code.

Figure 5.17
Byte code of the function `{3 + 5}`.

Figure 5.18
Simple function with arguments.

Figure 5.19
Using … for undefined number of arguments.

Figure 5.20
Using and overriding default values of arguments.

Figure 5.21
Performing messages chosen by index.

Figure 5.22
Evaluating functions chosen by index.

Figure 5.23
Asynchronous communication with a Server.

Figure 5.24
`loop` and the use of Event—`(key:value).play`—to play notes.

Figure 5.25
Partial application.

Figure 5.26
Iterative factorial.

Figure 5.27
Recursive factorial.

Figure 5.28
Recursion over a tree of unknown structure.

Figure 5.29
A function that creates functions that count.

Figure 5.30
Functions created by functions as models of instances.

Figure 5.31
Functions stored in events as instance methods.

Figure 5.32
Building an Array with add.

Figure 5.33
Summary of class definition parts (excerpt from the definition of a class node).

Figure 5.34
Classes of classes.

Figure 5.35
`Class` and `Meta_Class` are mutual instances of each other.

Figure 5.36
Counter class.

Figure 5.37
SynthDefs for the Counter model example.

Figure 5.38
A dependant that plays sounds.

Figure 5.39
A dependant that displays the count.

Figure 6.1
Example of a SynthDef.

Figure 6.2
Example of a key/value `Array` and a note `Event`.

Figure 6.3
Defining a `Pattern`, creating a Stream and extracting its values.

Figure 6.4
Two ways of writing the same Event pattern.

Figure 6.5
A more elaborate Event pattern.

Figure 6.6
Using Event types

Figure 6.7
Chord Events.

Figure 6.8
Interdependent key values in a Pattern.

Figure 6.9
Chaining `Event` patterns.

Figure 6.10
Using `Prout` to define and play patterns on the fly.

Figure 6.11
Using `Prout` to define value and event patterns.

Figure 6.12
Rendering and playing a pattern.

Figure 6.13
Sound-file granulation with a `Pattern`.

Figure 6.14
`Yield` versus `EmbedInStream`.

Figure 6.15
The definition of the stream created by `Pseq`.

Figure 6.16
The definition of `Event's` play method.

Figure 6.17
Definition of the key `\play` in the default event.

Figure 6.18
Implementation of the event type `\bus`.

Figure 7.1
A modulo algorithm that operates over states of variables.

Figure 7.2
Synthesis graph.

Figure 7.3
Dynamic synthesis graph.

Figure 7.4
Refactoring a synthesis graph at runtime.

Figure 7.5

Figure 7.6
A dynamic graph of a chaotic linear congruence.

Figure 7.7
Creating a Proxy object explicitly and changing its source.

Figure 7.8
Unified creation and access syntax with `Ndef`.

Figure 7.9
Unified creation and access syntax within an environment.

Figure 7.10
Initialization of node proxies in the proxy space.

Figure 7.11
Parameter mapping and setting.

Figure 7.12
Rewriting a synth def and a task def while running.

Figure 7.13
Embedding and forking of different tasks.

Figure 7.14
Passing an environment into a task proxy when embedding.

Figure 7.15
A pattern proxy as an entry point into a stream.

Figure 7.16
Deriving variations fom nonexisting streams by mathematical operations.

Figure 7.17
`Pdef`—play, pause, and resume.

Figure 7.18
A larger combination of `Pdefs`.

Figure 7.19
Simplifying the code in figure 7.18 using `Psym`.

Figure 7.20
Using `Pdefn` for the sequence of symbols itself.

Figure 7.21
`Event` type “phrase”.

Figure 7.22
Recursive phrasing.

Figure 7.23

Figure 7.24

Figure 7.25
Combinations between patterns and UGen graphs.

Figure 7.26
The same functionality, using `Ndef` instead of `ProxySpace`.

Figure 7.27
Using `Tdef` to create overlapping synths within a node proxy.

Figure 7.28
Distributed live coding with the History class.

Figure 7.29
Shared variables between remote proxy spaces.

Figure 8.1
A `Puppet` class, and some tests for it.

Figure 8.2
A puppet modeled as an event

Figure 8.3
Add more instance variables and change the blip method.

Figure 8.4
A minimal shout window sketch.

Figure 8.5
Wrap the sketch in a pseudomethod

Figure 8.6
More pseudomethods.

Figure 8.7
Text color animation.

Figure 8.8
A screenshot of the shout window so far.

Figure 8.9
Using `codeDump` to shout.

Figure 8.10
Updated `setMessage` to scale font and flash text.

Figure 8.11
A Shout class.

Figure 8.12
More class variables and an `initClass` method.

Figure 8.13
Adding `*makeWindow` and `*setMessage` to `Shout`.

Figure 8.14
Converting `z.shout` to `Shout.new`.

Figure 8.15
Converting `animate` to a class method.

Figure 8.16
Converting `setMessage`.

Figure 8.17
Handling `codeDump` in object model and class.

Figure 8.18
A fixed serialization method.

Figure 8.19
Flexible serialization by lookup.

Figure 8.20
Some initial serialization methods.

Figure 8.21
Adding a new serialization type at runtime.

Figure 8.22
Some granular SynthDefs and tests.

Figure 8.23
Global setup and a player `Tdef` for the cloud.

Figure 8.24
Tests for the cloud.

Figure 8.25
Making random settings and eight random presets to switch from one to the other.

Figure 8.26
Cross-fading between different settings with a `TaskProxy`.

Figure 8.27
A screenshot of the `CloudGenMini` GUI //

Figure 8.28
A lightweight GUI for `CloudGenMini`.

Figure 9.1
SuperCollider IDE on MacOS.

Figure 9.2
View of status bar in the IDE.

Figure 9.3
Starting up server through the status bar.

Figure 9.4
Automatic code completion in SuperCollider.

Figure 9.5
Method call assistance in the SuperCollider IDE.

Figure 9.6
SuperCollider IDE Configuration window.

Figure 10.1
scnvim extension for SuperCollider in Nvim.

Figure 10.2
scvsc and SuperCollider in VS Code.

Figure 10.3
User-defined keystrokes for sclang mode.

Figure 10.4
User-defined functions for sclang mode.

Figure 10.5
Code insertion example of user defined function for sclang mode.

Figure 10.6
Other approaches to user-defined functions.

Figure 10.7
Environment configuration for doom-load-envvars-file.

Figure 10.8
Installing scel package with the recipes method.

Figure 10.9
Interaction with SuperCollider in Tidal Cycles.

Figure 10.10
Active code block interaction in Babel and Org mode.

Figure 11.1
The four boards tested. Clockwise from top left: Beaglebone Black, NanoPi Neo, Raspberry Pi 2 B v1.1, and Raspberry Pi 4 B.

Figure 11.2
The ARM SBCs with audio connections and add-ons. Clockwise from top left: NanoPi Neo, with wiring that might offend hi-fi enthusiasts despite impressive audio quality metrics; Bela, mounted on Beaglebone with connectors; HiFiBerry DAC+ ADC Pro board; and HiFiBerry mounted on Pi 4.

Figure 11.3
The SuperCollider stack. sclang talks to scsynth, scsynth talks to jackd (and sclang), jackd talks to ALSA, ALSA talks to the audio hardware. Bela replaces jackd and ALSA with a custom audio driver setup.

Figure 11.4
The output of aplay -l on a Raspberry Pi 4 with a HifiBerry hat attached. Card 0 is the built-in audio, cards 1 and 2 are the double HDMI ports, and card 3 is the HifiBerry. The card number is important, as that is the device number to pass to jackd.

Figure 11.5
The command line mixer alsamixer, controlling a HifiBerry hat.

Figure 11.6
How many sine synths each ARM SBC can run at a sample rate of 44,100 Hz and various buffer sizes.

Figure 11.7
How many FFT synths each ARM SBC can run at a sample rate of 44,100 Hz and various buffer sizes.

Figure 11.8
The $150 effects pedalboard, showing a mock-up of four NanoPi Neos that are connected.

Figure 11.9
Live coding with live percussion setup. The live percussion is processed through a low-latency Bela setup, which is remotely controlled via OSC messages emitted from a live coding environment.

Figure 12.1
Basic creation and manipulation of a window.

Figure 12.2
Practical usage of `closeAll` and `screenBounds`.

Figure 12.3
Creation and placement of a `Slider` view in a parent window.

Figure 12.4
Usage of auto-layout tools

Figure 12.5
Generalized syntax for getting and setting.

Figure 12.6
Getting and setting attributes of a slider.

Figure 12.7
Controlling signal amplitude with a slider.

Figure 12.8
Controlling signal amplitude with a button.

Figure 12.9
Using `lincurve` to map amplitude values nonlinearly.

Figure 12.10
Using `ControlSpec` to map frequency values.

Figure 12.11
Controlling a frequency parameter using EZSlider.

Figure 12.12
Using subviews to create and organize hierarchical sections of a GUI.

Figure 12.13
Defining custom actions to be invoked in response to mouse/keyboard input.

Figure 12.14
Controlling a view with MIDI pitch bend data.

Figure 12.15
Scheduling GUI updates with a `Routine`.

Figure 12.16
Basic usage of `Pen` and `UserView`.

Figure 12.17
A custom button with `UserView` and `Pen`.

Figure 12.18
Creating an animated effect with `UserView` and `Pen`.

Figure 12.19
An envelope GUI that uses a `dependancy` structure.

Figure 12.20
Using `SkipJack` to update a GUI element.

Figure 13.1
Overview of a dataset on the number of publications and their citations on the topic "sonification, auditory display, audification, sonify" (first 1000 hits of a Google search from 2023; (Groß-Vogt et al., 2023)). (a) shows the number of papers published per year. (b) is a boxplot of the number of citations per year.

Figure 13.2
ASA—example of detuning one partial of a harmonic sound.

Figure 13.3
Basic audification of an arbitrary function

Figure 13.4
A basic auditory graph.

Figure 13.5
Basic example of model-based sonification.

Figure 13.6
Focused audification with an arbitrary example data set.

Figure 13.7
Loading the data.

Figure 13.8
Mapping the data to pitch in discrete events.

Figure 13.9
Continuous data sonification.

Figure 13.10
Sound design with noise pulses.

Figure 13.11
Four regions in sequence.

Figure 13.12
Sonification of tag systems.

Figure 13.13
Example for Dark Matter BEER approach.

Figure 13.14
Gravitational model.

Figure 13.15
Sonify world objects and forces.

Figure 13.16
Sound design for a single country.

Figure 13.17
Terra Nullius: moving latitude zone on the globe.

Figure 14.1
Mono and multiple mono.

Figure 14.2
Common multichannel configurations, from left to right: stereo, quadrophonic, octophonic, and 5.1.

Figure 14.3
Overview of the different panners in SuperCollider with their argument names.

Figure 14.4
Visualization of the arguments of `PanAz`.

Figure 14.5
Delay lines and their arguments.

Figure 14.6
`Tap` example.

Figure 14.7
The `Convolution` UGens and their argument.

Figure 14.8
The `Reverb` UGens with their input arguments.

Figure 14.9
Binaural audio engine.

Figure 14.10
2D and 3D VBAP `Speaker Arrays`.

Figure 14.11
3D VBAP example.

Figure 14.12
The Ambisonic UGens in the main SuperCollider distribution.

Figure 14.13
Huygens principle.

Figure 14.14
WFS.

Figure 14.15
WFS speakers for the the Game of Life system (August 2007).

Figure 14.16
Example of WFS calculation from the WFS-lib from the `WFSPan` class by Wouter Snoei.

Figure 14.17
An example of the use of `PV_Decorrelate`.

Figure 14.18
Spatial diffusion through granulation.

Figure 14.19
Simple spectral diffusion example.

Figure 15.1
Immediate machine listening example using `Pitch` and `Amplitude` UGens. The original detected pitch appears in your left ear and an octave higher in the right one.

Figure 15.2
`Loudness`.

Figure 15.3
`MFCC`.

Figure 15.4
`Onsets`.

Figure 15.5
`BeatTrack`.

Figure 15.6
`KeyTrack`.

Figure 15.7
Simple melodic transcription.

Figure 15.8
`OnlineMIDI`.

Figure 16.1
Short grain durations—pitch to colored click.

Figure 16.2
Perception of short silences.

Figure 16.3
Order confusion with sounds in fast succession.

Figure 16.4
Multiple grains fuse into one composite.

Figure 16.5
Plot of envelope, waveform, and grain.

Figure 16.6
Making different envelope shapes.

Figure 16.7
A sinc function envelope as array.

Figure 16.8
More envelopes plotted.

Figure 16.9
`SynthDefs` with different envelopes.

Figure 16.10
Changing grain duration, frequency, and envelope.

Figure 16.11
Different control strategies applied to density.

Figure 16.12
Control strategies applied to different parameters.

Figure 16.13
`GrainFM` with individual control proxies.

Figure 16.14
`GrainBuf` and control proxies.

Figure 16.15
Glisson synthesis.

Figure 16.16
Pulsar basics—making a set of waveform and control tables.

Figure 16.17
Pulsars as node proxies using `GrainBuf`.

Figure 16.18
Making new tables and send them to buffers.

Figure 16.19
A screenshot of the nuPG program.

Figure 16.20
The sieve GUI of the nuPG program.

Figure 16.21
Making, combining, and playing sieves.

Figure 16.22
Time-pitch changing.

Figure 16.23
A constant-Q `SynthDef` and `Pattern`.

Figure 16.24
A Wavesets object.

Figure 16.25
A single waveset and a waveset group.

Figure 16.26
Wavesets, buffers, and events.

Figure 16.27
A pattern to play wavesets.

Figure 16.28
Some of Wishart's transforms.

Figure 16.29
Waveset substitution.

Figure 16.30
Wavesets played with `Tdef`.

Figure 16.31
A wait time derived from waveset duration, and a gap added.

Figure 16.32
Add pitch contour and dropout rate.

Figure 17.1
Example of `PlayBuf` with `midiratio`.

Figure 17.2
Example of `Pbind` with `\midinote`.

Figure 17.3
Example of `Pbind` with `\midinote` and cents.

Figure 17.4
Example of `Pbind` with `\note`.

Figure 17.5
Example of `Pbind` with `\degree`.

Figure 17.6
Example of `Pbind` with `\degree` and `\scale` (chromatic scale).

Figure 17.7
Example of `Pbind` with `\degree` and `\scale` and cents.

Figure 17.8
Example of `Pbind` with `\stepsPerOctave`.

Figure 17.9
Example of `Pbind` with `\stepsPerOctave` and `\scale`.

Figure 17.10
Example of changing mode using `\scale`.

Figure 17.11
Example of a simple tuning of three equal notes per octave.

Figure 17.12
Example of `Pbind` with unequal octave divisions for `\freq`.

Figure 17.13
Example of using odd-limit ratios with sound.

Figure 17.14
Code for a tonality diamond with a SuperCollider GUI.

Figure 17.15
Picture of a tonality diamond with the SuperCollider GUI.

Figure 17.16
Example of 12 and 14 equal notes per octave mixed together.

Figure 17.17
Example of two tunings organized by their most common notes.

Figure 17.18
Example of calculating near-tones, setting a tolerance threshold.

Figure 17.19
Example changing the number of common notes in real-time.

Figure 17.20
Example of 12 equal-note division beyond the octave.

Figure 17.21
Example of Tuning by W. Carlos with SuperCollider.

Figure 17.22
Example of `Pbind` with unequal divisions below an octave.

Figure 17.23
Example of a pattern-based tuning.

Figure 17.24
Calculation of the first 16 harmonics for a root note of 440 Hertz.

Figure 17.25
Rearranging the first 16 harmonics within one octave.

Figure 17.26
Different arrangements for the first 16 harmonics.

Figure 17.27
Creating virtual partials for a seven equal-note tuning.

Figure 17.28
Code for a simple GUI keyboard to trigger the seven equal-note temperament.

Figure 17.29
Picture of a simple GUI keyboard to trigger the seven equal-note temperament.

Figure 18.1
Basic steps to save binary OSC messages for later use offline.

Figure 18.2
Example of `Score` usage.

Figure 18.3
Build relationships with variables and functions rather than bard-coding values.

Figure 18.4
Fill a `Score` algorithmically.

Figure 18.5
Memory allocation within `Score` for NRT.

Figure 18.6
Memory allocation outside of `Score` for real-time usage.

Figure 18.7
Example real-time `CtkScore` usage.

Figure 18.8
Fill and modify `CtkScore` algorithmically.

Figure 18.9
Build a granular gesture with Ctk for offline rendering algorithmically.

Figure 18.10
Example of non-real-time Ctk usage.

Figure 18.11
VSO_Vib class definition.

Figure 18.12
`VSO_ADR` class definition.

Figure 18.13
VSO initialization method.

Figure 18.14
Complete VSO class definition.

Figure 18.15
Example of `CtkScore` from figure 18.10 with VSO substitution.

Figure 18.16
Complete `NRT_TimeFrame` class definition.

Figure 18.17
Example of `CtkScore` from figure 18.15 using `NRT_TimeFrame`.

Figure 19.1
Code for a 500-step symmetric random walk.

Figure 19.2
Picture of a 500-step symmetric random walk.

Figure 19.3
Code for one hundred 500-step symmetric random walks.

Figure 19.4
Picture of one hundred 500-step symmetric random walks.

Figure 19.5
An asymmetric random walk with a pair of elastic barriers.

Figure 19.6
`Pbrown`.

Figure 19.7
Random walks over the pitch sieve of Jonchaies.

Figure 19.8
The melody of Steve Reich's Piano Phase (1967), shifting out of phase.

Figure 19.9
Sequences of random walks.

Figure 19.10
DSS: two cycles of a waveform with three breakpoints.

Figure 19.11
`DemandEnvGen`.

Figure 19.12
DSS employing `Dbrown` and `Dswitch1` UGens.

Figure 19.13
Code for a GUI for DSS.

Figure 19.14
Picture of a GUI for DSS.

Figure 19.15
`Pseg`.

Figure 19.16
Tendency masks for pitch, panning, and duration.

Figure 19.17
Tendency mask for pitch values of the first example from figure 19.16.

Figure 19.18
Tendency masks for the concatenation of waveforms.

Figure 19.19
Example by Viznut.

Figure 19.20
Code that makes a `UGen` graph `Function`.

Figure 19.21
Variants of the “42 Melody,” cited by Viznut.

Figure 19.22
Alternatives to an integer counter.

Figure 19.23
Examples with the greatest common divisor and least common multiple.

Figure 19.24
Basic bit scrambling by passing permutation arrays.

Figure 19.25
Dynamic bit mapping

Figure 19.26
Equivalence of buffer modulation with `BufRd` and `PlayBuf`.

Figure 19.27
Global and local movement.

Figure 19.28
Variants of global and local movement.

Figure 19.29
Model of main modulation and disturbance.

Figure 19.30
Modulation with `DemandEnvGen`.

Figure 19.31
Basic changes of writing and reading rates.

Figure 19.32
Rewriting with variable buffer length and GUI control.

Figure 19.33
Buffer modulation and rewriting combined.

Figure 19.34
Buffer rewriting with feedback.

Figure 19.35
Buffer rewriting with overdubbing.

Figure 20.1
Simple find-replace keyword substitution

Figure 20.2
Regular expressions for keyword searching.

Figure 20.3
Lack of context in string substitution.

Figure 20.4
String replacement, with awareness of string-literal context.

Figure 20.5
A preprocessor of f-string.

Figure 20.6
F-string preprocessor test suite.

Figure 20.7
Tree structures derived from code statements.

Figure 20.8
Abstract class for parse-tree nodes.

Figure 20.9
Parsing node for f-strings.

Figure 20.10
Preprocessor for f-string parsing classes.

Figure 20.11
Mini-dialect rhythmic notation.

Figure 20.12
Preprocessor for the bracket-group dialect.

Figure 20.13
Usage example of the pattern preprocessor.

Figure 21.1
A screenshot of ixiQuarks.

Figure 21.2
A `Synth` definition with 10 harmonics.

Figure 21.3
Code with horizontal sliders to control the frequency and amplitude of our synth.

Figure 21.4
A screenshot of the GUI with horizontal sliders.

Figure 21.5
Code with vertical sliders to control the frequency and amplitude of our synth.

Figure 21.6
A screenshot of the GUI with vertical sliders.

Figure 21.7
Irritia—A stochastic patch playing with the envelope view; the mouse can be used to interact with the patch.

Figure 21.8
A screenshot of ParaSpace.

Figure 21.9
In the Gridder, we see how the `Grid` view is used to map microtonal scales (the horizontal columns) to the octaves (the vertical rows). One can then play the Gridder with the mouse or a pen tablet as if the lines of active nodes were strings. Agents can be set to move in the system and trigger automatic performance. We also see how the `MIDIKeyboard` view is used to show the pitch of the node on the grid. (It is gray if it fits the equal tempered scale, but red if it is a microtone.)

Figure 21.10
In the PolyMachine, the `BoxGrid` is used for polyrhythmic step sequencers. It can have any number of columns and rows, but here, it has only one row for each track and a row above it, for indicating which step the clock is at.

Figure 21.11
A screenshot of the Shooting Scales performance instrument.

Figure 21.12
Snjókorn—a patch with four layers of `MultiSliderView` triggering sounds.

Figure 21.13
A screenshot of sounddrops in microtonal mode, in which each of the drops (ranging

from 2 to 48 in the view) has properties such as sound function (sample, synthesis, code, or audiostream), pitch, amplitude, speed, and steps. The microtonal keyboard under the multislider view has seven octaves (vertical) and from 5 to 48 notes (equal tempered tuning) per octave.

Figure 21.14
Shown here is sounddrops in tonal mode. There are two pop-up windows as well: one is the coding window for live coding, and the other is a window that contains various scales and chords that can be used to color the keyboard. These keys can then be used to assign the frequencies to the drops.

Figure 22.1
SuperCollider socks.

Figure 22.2
Top page of http://supercollider.jp, consisting of a wiki, forums, and more information.

Figure 22.3
Playing SC on an iPhone.

Figure 23.1
The ENIAC Cycling Unit and the structure of the 10 pulses that it creates.

Figure 23.2
Two ways of constraining parameters.

Figure 23.3
Overview of the registers for one `Pokey` channel.

Figure 23.4
Modulating Pokey inputs.

Figure 23.5
Block SuperCollider “coolant” example.

Figure 23.6
`HierSch` scheduling constraints and priority levels.

Figure 23.7
Priority-based `HierSch` scheduling.

Figure 23.8
`Maybe` yes.

Figure 23.9
Coprimes as frequency and trigger rates.

Figure 23.10
A very simple notation translator.

Figure 24.1
Live performance of Ultraorbism by Marcel.li Antúnez, featuring music by Andrea Valle, La Factoria d'Arts Escèniques de Banyoles. Banyoles, Spain. November 16, 2019. Photograph by Carles Rodríguez, 2019.

Figure 24.2
“Symbolic Machines” by Marianne Teixidó at Pumpumyachkan Asimtria 17. Cusco, Peru, 2021.

Figure 24.3
Screenshot from the LNX_Studio software.

Figure 24.4
Hidden Encounters sound installation, Ann Warde.

Figure 25.1
Loading data from two CSV files and training two `PPMC` models.

Figure 25.2
Generating values for pitch and IOI and playing them in a loop.

Figure 25.3
Extract feature data.

Figure 25.4
Generate and train a `SARDNET` instance, then test each second of the song against the SARDNET map.

Figure 25.5
Calculate distances between the vectors and record according to similarity.

Figure 25.6
Use the mouse pointer to navigate the track along the x-axis of the screen.

Figure 25.7
Initialization code for sound classification.

Figure 25.8
Collection of training data.

Figure 25.9
Training the neural network.

Figure 25.10
Classifying live sound with the trained neural network.

Figure 25.11
Code for initializing the neural network and defining a phase modulation synthesizer.

Figure 25.12
Collection of training data.

Figure 25.13
Training the neural network.

Figure 25.14
Using the trained model for mapping mouse movements into synthesis parameters.

Figure 25.15
A synth that captures mouse coordinates, sends them to the model, and receives sound spectra back from the model.

Figure 25.16
OSC communication between SuperCollider and the VAE model.

Figure 26.1
`WinBlock` used with `MITHScreenRatios`.

Figure 26.2
`MITHNoteViewer` dynamic note array presentation with `Synth`.

Figure 26.3
Screenshot of `MITHNoteViewer` as shown in figure 26.2.

Figure 26.4
Simple `GMITH` score creation and galley view display.

Figure 26.5
Example of `GMITH` score view using makePage method.

Figure 26.6
Dynamic CPWN using a function to generate an algorithmic chord sequence.

Figure 26.7
Dynamic score with audio.

Figure 26.8
Screenshot for `INScore` scene scaling, object and dynamic CPWN animation.

Figure 26.9
Screenshot taken from the “metals” section of Unthinking Things.

Figure 26.10
A stone “constellation” from Unthinking Things.

Figure 26.11
Other notations including CPWN from Unthinking Things. (a) text and CPWN (“melismata”)27; (b) “vocal rhythms”28.

Figure 26.12
Display notes on a staff.

Figure 26.13
Result of code in figure 26.12.

Figure 26.14
Attach articulation and dynamics marks to a note.

Figure 26.15
Result of code in figure 26.14.

Figure 26.16
Attach slur and dynamic hairpin spanners to a selection of notes.

Figure 26.17
Result of code in figure 26.16.

Figure 26.18
Attach a LilyPond literal string to the first note in a selection.

Figure 26.19
Result of code in figure 26.18.

Figure 26.20
The `transpose` and `scale` mutation methods.

Figure 26.21
Results of code in figure 26.20.

Figure 26.22
Provide a custom SynthDef for playing a FoscStaff.

Figure 26.23
Simple usage of `FoscMusicMaker`.

Figure 26.24
Result of code in figure 26.23.

Figure 26.25
`FoscMusicMaker` used to create divisive rhythmic structures.

Figure 26.26
Results of code in figure 26.25.

Figure 26.27
Melodic sequences demonstrating the use of a mask

Figure 26.28
Results of code in figure 26.27.

Figure 26.29
A customized extension of `FoscMusicMaker`: `InconjunctionsMaker`.

Figure 26.30
Results of code in figure 26.29.

Figure 27.1
Syntax techniques used in SCTweets.

Figure 27.2
“Scottish Raga” by Nathaniel Virgo.

Figure 27.3
Recursive modulation of frequency, phase, and amplitude, by Fredrik Olofsson.

Figure 27.4
Analytical expressions of Olofsson’s recursive SCTweet.

Figure 27.5
Quine, by Nathan Ho.

Figure 28.1
Overview of the SuperCollider language subsystems.

Figure 28.2
`PyrSlot` data structure.

Figure 28.3
32 bit `PyrSlot` memory layout.

Figure 28.4
`PyrObjectHdr` data structure.

Figure 28.5
Object memory layout.

Figure 28.6
`PyrSymbol` data structure.

Figure 28.7
`VMGlobals` data structure.

Figure 28.8
`ClassDependancy` data structure.

Figure 28.9
`PyrBlock` data structure.

Figure 28.10
`PyrMethodRaw` data structure.

Figure 28.11
`PyrMethod` data structure.

Figure 28.12
Stack layout for message sends and primitive calls.

Figure 28.13
`PyrFrame` data structure.

Figure 28.14
Activation frame call chain.

Figure 28.15
Most important `Interpreter` implementation entities (some details omitted).

Figure 28.16
Implementation of an example primitive for `atan2`.

Figure 29.1
`Reverb1`.

Figure 29.2
`Reverb1` example.

Figure 29.3
`Reverb1` additions.

Figure 29.4
`SinOsc` example.

Figure 29.5
Flanger.

Figure 29.6
`checkInputs`.

Figure 29.7
`checkSameRateAsFirstInput`.

Figure 29.8
`Flanger` UGen first version, in C code.

Figure 29.9
`Flanger` in action.

Figure 29.10
`Flanger` UGen second version, C code.

Figure 29.11
Choosing a calculation function.

Figure 29.12
Control rate trigger input.

Figure 29.13
Random number generator.

Figure 29.14
`MultiOutUGen`.

Figure 29.15
`RecordBuf`.

Figure 29.16
Autovectorization.

Figure 30.1
Class diagram of significant domain entities.

Figure 30.2
Real-time threading and messaging implementation structure.

Figure 30.3
Real-time thread and queue instances and asynchronous message channels.

Figure 30.4
Overview of multithreaded processing of the `/b_allocRead` command.

Figure 30.5
Stage 1 of processing the `/b_allocRead` command in the real-time context.

Figure 30.6
Stage 2 of processing the `/b_allocRead` command in the NRT context.

Figure 30.7
Stage 3 of processing the `/b_allocRead` command in the real-time context.

Figure 30.8
Stage 4 of processing the `/b_allocRead` command in the NRT (NRT) context.

List of Tables

Table 1.1
Logical expressions

Table 1.2
Scale and offset examples

Table 4.1
MIDI messages and their corresponding implementations in SuperCollider

Table 6.1
`Event` types of the default event

Table 6.2
`Event` timing keys

Table 6.3
Command transmission keys

Table 6.4
Node-related keys

Table 6.5
Amplitude-related keys

Table 6.6
Note articulation-related keys

Table 6.7
Frequency-related keys

Table 6.8
Numerical pattern classes

Table 6.9
`Event` pattern classes

Table 6.10
`Pattern` pattern classes

Table 10.1
Popular alternative IDEs for SuperCollider, with cross-platform links for installation of the basic IDE and SuperCollider specific extension

Table 10.2
Key bindings in sclang mode

Table 11.1
Overview of features for tested ARM SBCs

Table 11.2
Results of the audio I/O quality tests

Table 11.3
Audio performance test results

Table 12.1
Names and descriptions of commonly used view subclasses

Table 12.2
A partial list of methods for mapping one value range to another

Table 12.3
Names and descriptions of commonly used EZ-GUI classes

Table 12.4
Commonly used class methods for `Pen`

Table 13.1
Mappings of Data to Sonification Parameters in “Navegar”

Table 15.1
Overview of machine listening resources for SuperCollider

Table 16.1
Overview of waveset transforms

Table 17.1
Example of Tunings by W. Carlos.

Table 20.1
Test cases of f-strings

Table 28.1
`PyrSlot` tags

Table 28.2
Object formats

Table 28.3
Object flags

Table 28.4
Parse node classes

Table 28.5
Virtual machine byte codes, listed along with the corresponding constants from *`Opcodes.h`* where appropriate

Table 28.6
Method type enumeration values

Table 28.7
`PyrSlot` access functions

Table 28.8
Language implementation headers and sources

Table 29.1
Useful macros for UGen writers, part I

Table 29.2
Memory allocation and freeing

Table 29.3
Useful Macros for UGen writers, part II

Table 30.1
Subclasses of `SequencedCommand` Defined in SC_SequencedCommand.cpp

Foreword

James McCartney

Why use a computer programming language for composing music? Specifically, why use SuperCollider? There are several very high-level language environments for audio besides SuperCollider, such as Common Music, Kyma, Nyquist, and Patchwork. These all demonstrate very interesting work in this area and are worth looking into. SuperCollider, though, has the unique combination of being free, well supported, and designed for real time. SuperCollider is a language for describing sound processes. SuperCollider is very good at allowing one to create nice sounds with minimal effort, but more importantly it allows one to represent musical concepts as objects, to transform them via functions or methods, to compose transformations into higher-level building blocks, and to design interactions for manipulating music in real time, from the top-level structure of a piece down to the level of the waveform. You can build up a library of classes and functions that become building blocks for your working style and in this way make a customized working environment. With SuperCollider, one can create many things: very long or infinitely long pieces, infinite variations of structure or surface detail, algorithmic mass production of synthesis voices, sonification of empirical data or mathematical formulas, to name a few. It has also been used as a vehicle for live coding and networked performances. It is because of this open-endedness that early on, I often felt it difficult to know how best to write the documentation. There were too many possible approaches and applications.

So, I am pleased that there will now be a book on SuperCollider, and the best part of it for me is that I have not had to do much of the hard work to get it done. Since I made SuperCollider open source, it has taken on a life of its own and become a community-sustained project as opposed to being a project sustained by a single author. Many people have stepped up and volunteered to undertake tasks of documentation, porting to other operating systems, interfacing to hardware and software, writing new unit generators, extending the class library, maintaining a website, mailing lists, a wiki, fixing bugs, and finally writing and editing the chapters of this book. All of these efforts have resulted in more features, better documentation, and a more complete, robust, and bug-free program.

SuperCollider came about as the latest in a series of software synthesis languages that I have written over the years. I have been interested in writing software to

synthesize electronic music ever since I was in high school. At the time, I had written on a piece of notebook paper a set of imaginary subroutine calls in BASIC for implementing all the common analog synthesizer modules. Of course, doing audio synthesis in BASIC on the hardware of that time was completely impractical, but the idea had become a goal of mine. When I graduated college, I went out to have a look at E-mu in California and found that they were operating out of a two-story house. I figured that the synthesizer industry was a lot smaller than I had imagined and that I should re-think my career plans.

The first software synthesizer I wrote was a graphical patching environment called Synfonix which generated samples for the Ensoniq Mirage sampling keyboard using a Macintosh computer. I attempted to sell this program but I only sold two copies, one to Ivan Tcherepnin at Harvard and another to Mark Polishook. A business lesson I learned from this is not to sell a product that requires purchasers to already own two niche products. The intersection of two small markets is near zero.

In 1990, I wrote a program called Synth-O-Matic that I used personally but never meant to distribute even though one copy I gave to a friend got circulated. I created this program after learning CSound and deciding that I never wanted to actually have to use CSound's assembly-language-like syntax. Synth-O-Matic had a more expression-oriented syntax for writing signal flow graphs and a graphical user interface for editing wave tables. I used this program on and off, but it was quite slow so I stopped using it for the most part in favor of hardware synthesizers.

It wasn't until the PowerPC came out that it became practical for the first time to do floating-point signal processing in real time on a personal computer. At the time I had been working on music for a modern dance piece using Synth-O-Matic to do granular synthesis. It was taking a long time to generate the sound and I was running behind schedule to get the piece done. On the day in March 1994 when the first PowerPC based machine came out I went and bought the fastest one, recompiled my code, and it ran 32 times faster. I was then able to complete the piece on time. I noticed that my code was now running faster than real time, so I began working on a program designed to do real-time synthesis. Around this time I got a note in the mail from Curtis Roads, who had apparently gotten one of the circulating copies of Synth-O-Matic, encouraging me to further develop the program. So, I took the Synth-O-Matic engine and combined it with the Pyrite scripting language object which I had written for MAX. This became SuperCollider version 1, which was released in March 1996. The first two orders were to John Bischoff and Chris Brown of The Hub.

The name "SuperCollider" has an amusing origin. During the early 1990s, I worked at the Astronomy Department of the University of Texas on the Hubble Space Telescope Astrometry Science Team where I was writing software for data analysis and telescope observation planning. On the floors below my office was the Physics Department, some

of the members of which were involved in the Superconducting Super Collider project. In 1993, Congress cut the funding for the project and there were many glum faces around the building after that. I had been thinking about this merging, or "collision" if you will, of a real-time synthesis engine with a high-level garbage collected language and it seemed to me that it was an experiment that would likely fail, so I named it after the failed Big Science project of the day. Except that the experiment didn't fail. To my surprise, it actually worked rather well.

The version 1 language was dynamically typed, with a C-like syntax, closures borrowed from Scheme, and a limited amount of polymorphism. After using version 1 for a couple of years, and especially after a project where I was invited by Iannis Zannos to work on a concert in an indoor swimming pool in Berlin, I realized that it had severe limitations in its ability to scale up to create large working environments. So I began working on version 2 which borrowed heavily from Smalltalk. It was for the most part the same as the language as described in this book but for the synthesis engine and class library, which have changed a great deal.

The goal of SuperCollider version 2 and beyond was to create a language for describing real-time interactive sound processes. I wanted to create a way to describe categories of sound processes that could be parameterized or customized. The main idea of SuperCollider is to algorithmically compose objects to create sound-generating processes. Unit generators, a concept invented by Max Matthews for his Music N languages, are very much like objects and are a natural fit for a purely object-oriented Smalltalk-like language.

In 2001, I was working on version 3 of SuperCollider, and because of the architecture of the server it was looking like it really should be open source so that anyone could modify it however they liked. I was (barely) making a living at the time selling SuperCollider so the decision to give it away was a difficult one to make. But financially it was looking like I would need a "real" job soon anyway. I was also worried that the period of self-employment on my résumé would begin looking suspect to potential employers. Ultimately, I did get a job, so I was able to open source the program. On the day that I made all of my previously copyprotected programs free, my website experienced an eightfold increase in traffic. So obviously, there was a lot of interest in an open-source audio language and engine, especially one that is free.

I hope that this book will enable and inspire the reader to apply this tool in useful and interesting ways. And I hope to be able to hear and enjoy the results!

Introduction to the Second Edition

Scott Wilson, David Cottle, and Nick Collins

Welcome to the second edition of *The SuperCollider Book*! It's been more than 10 years since the first edition, and we're delighted to present a refreshed and expanded collection of tutorials, essays, and projects that highlight one of the most exciting and powerful of all audio environments. For this new edition, we've thoroughly updated the existing chapters, reworked some topics completely, and added new content to bring the book up-to-date with the most exciting aspects of SuperCollider practice today.

SuperCollider (SC to its friends) is a domain-specific programming language, specialized for sound but with capabilities to rival any general-purpose language. While it is technically a blend of Smalltalk, C, and ideas from a number of other programming languages, many users simply accept SuperCollider as its own wonderful dialect, a superlative tool for real-time audio adventures. Indeed, for many artists, SuperCollider is the first programming language they learn, and they do so without fear because the results are immediately engaging; you can learn a little at a time in such a way that you hardly notice that you're programming until it's too late—and you're hooked! The potential applications in real-time interaction, installations, electroacoustic pieces, generative music, audiovisuals, and a host of other possibilities make up for any other qualms. On top of that it's free, powerful, open source, and has one of the most supportive and diverse user and developer communities around.

1 Pathways

This book will be your companion to SuperCollider; some of you will already have experience and be itching to turn to the many varied chapters further on from this point. We'll let you follow the book in any order you choose! But we would like to take care to welcome any newcomers and point them straight in the direction of chapter 1, which provides a friendly introduction to the basics of the language itself. Most of you will probably be learning SC using the dedicated SuperCollider IDE (Integrated Development Environment), in which case reading chapter 9 in tandem may provide some useful information on its particularities. If you're using one of the more common alternative editors, such as Emacs, chapter 10 may prove similarly useful. From there, we suggest beginners continue through the early part of the book until chapter 4, as this

path will provide you with some basic skills and knowledge which can serve as a foundation for further learning.

For more advanced users, we suggest you look at the more topics-oriented chapters which follow. These chapters aren't designed to be read in any particular order, so proceed with those of particular interest and relevance to you and your pursuits. Naturally, we have referred to other chapters for clarification where necessary and have tried to avoid duplication of material except where absolutely crucial for clarity.

These topics chapters are divided into parts called "Advanced Tutorials," "Editors and GUI," and "Practical Applications." They begin with chapter 5, which provides a detailed overview of SuperCollider as a programming language. This may be of interest to beginners with a computer science background, who'd rather approach SC from a language design and theory perspective than through the more user-friendly approach given in chapter 1. Chapters on a variety of subjects follow, including sonification, spatialization, microsound, nonstandard sound synthesis, GUIs, machine listening, alternative tunings, SC on small computers, and non-real-time synthesis.

Following these chapters is a part designed for both intermediate and advanced users entitled "Projects and Perspectives." The material therein provides examples of how SuperCollider has been used in the real world. These chapters also provide some philosophical insight into issues of language design and its implications (most specifically in the case of chapter 23). This sort of intellectual pursuit has been an important part of SuperCollider's development; it is a language that self-consciously aims for good design and to allow and encourage elegance, and even beauty, in users' code. Although this might seem a little abstract at first, we feel that this sets SC apart from other computer music environments, and as users advance, awareness of such things can improve their code. Chapter 27, on SCTweets, provides a glimpse into one alternative practice in SC programming, in which compact code and special coding tricks allow surprisingly varied sonic output. One of the new contributions that we're most excited about is a chapter made up of personal artists' statements from a wide range of backgrounds, which give useful insights into the many different ways that SuperCollider is used.

Finally, there is a part entitled "Developer Topics," which provides detailed "under the hood" information about SC. These chapters are for advanced users seeking a deeper understanding of SC and its workings, and to those wishing to extend SC, for instance by writing custom unit generator plug-ins.

2 Code Examples and Text Conventions

Initially, SuperCollider was Mac only, but as an open-source project since 2001, it has widened its scope to cover all major platforms and a variety of editors, with only minor

differences between them. Most code in this book should run everywhere with the same results, and we will note places where there are different mechanisms in place. However, there are some differences in the programming editor environments (such as available menu items) and the keyboard shortcuts. You are referred as well to the extensive help system that comes with the SuperCollider application; a help file on Keyboard Shortcuts for the various editors/platforms can be found by searching in the documentation.

Just to note, when you come across keyboard shortcuts in the text they'll appear like this: [enter] designates the "enter" key, [ctrl+a] means the control key plus the "a" key, and so on. Furthermore, all text appearing in the `code font` will almost always be valid SuperCollider code. (Very occasionally, there may be exceptions for didactic purposes, like here.) Note that some code examples are line wrapped in the book due to layout limitations, but not in the code files available on the website. This should usually be visible as a slight indent. You will also encounter some special SuperCollider terms (like Synth, SynthDef, and Array) that aren't in code font and discussed in a friendly manner; this is because they are ubiquitous concepts, and it would be exhausting to have them in the code font every time. Anyway, if you're new to SuperCollider, don't worry about this at all; chapter 1 will start you on the righteous path, and you'll soon be chatting about Synths like the rest of us.

3 The Book Website

This brings us to the accompanying website for the book, which contains all the code reproduced within, ready to run, as well as download links to the application itself, its source code, and all sorts of third-party extras, extensions, libraries, and examples. At the time of publication, this is hosted at https://github.com/supercollider/scbookcode. Should it ever move, we promise to always provide an updated link from the main SC website. A standardized version of SuperCollider is used for the book, SuperCollider 3.13.0, for which all the book code should work without trouble. Of course, the reader may find it productive to download newer versions of SuperCollider as they become available, and it is our intention to provide updated versions of the example code where needed. Although we can make no hard-promises, in this fast-paced world of operating system shifts, that the code in this book will remain eternally correct—the ongoing development and improvement of environments like SuperCollider are a big part of what makes them so exciting—we've in any case done our best to present you with a snapshot of SuperCollider that should retain a core validity in future years.

4 Final Thoughts

Please be careful with audio examples; there is of course the potential to make noises that can damage your hearing if you're not sensible with volume levels. Until you become accustomed to the program, we suggest you start each example with the volume all the way down, and then slowly raise it to a comfortable level. (And if you're not getting any sound, remember to check if you've left the monitors off or the computer audio muted.) Some examples may use audio input, and have the potential to feedback unless your monitoring arrangements are correct. The easiest way to deal with such examples is to monitor via headphones.

We couldn't possibly cover everything concerning SuperCollider, and there are many online resources to track down new developments and alternative viewpoints, including mailing lists, forums, and a host of artists, composers, technology developers, and SuperCollider maniacs with interesting pages. We have provided a few of the most important links below, at the time of writing, but WikigoopediGPTle, or whatever your contemporary equivalent is, will allow you to search out the current SuperCollider 15.7 as necessary.

We're sure you'll have fun as you explore this compendium, and we're also sure you'll be inspired to some fantastic art and science as you go. Enjoy exploring the SuperCollider world first charted by James McCartney, but since expanded immeasurably by everyone who partakes in this infinitely flexible open-source project.

5 Primary Web Resources

Main community homepage:
http://supercollider.github.io/
James McCartney's home page:
http://www.audiosynth.com/
The sccode examples site:
https://sccode.org
The SC forum:
https://scsynth.org/

6 Acknowledgments

We owe a debt of gratitude to the chapter contributors, and the wider SuperCollider community, who have supported this project. SC's community is one of its greatest strengths, and innumerable phone calls, emails, chats over cups of tea at SC Symposia, Slack check-ins etc., have contributed to making this book and SC itself stronger. We apologize to all who have put up with our insistent editing, thank the chapter authors for their amazing content both new and revised, and acknowledge the great efforts of the developers in making SC even better over the years since the first edition. We would like to offer special thanks to Josh Parmenter for his assistance in carefully updating the developer chapters to reflect the many changes in the codebase since 2011. A thousand thank-yous:

```
1000.do{"domo arigato gozaimashita".postln};
```

Thanks to all at the MIT Press who have assisted in the handling of our proposal and manuscript for this book.

Thank you to friends, family, and colleagues who had to deal with us while we were immersed in the lengthy task of organizing, editing, and assembling this book—again!. Editing may seem like a solitary business, but from their perspective we're quite sure it was a team effort!

Many thanks to our students, who have served as guinea pigs for pedagogical approaches, tutorial materials, experimental developments, and harebrained ideas. Now back to your exercise wheels!

Finally, the editors would like to thank each other for support during the period of gestation. At the time of writing, we've been working for almost two years to bring the second-edition manuscript to fruition. We're older now, and while hopefully wiser, that hasn't made this any less of a labor of love the second time around, especially bringing it all together in the midst of recovering from the pandemic, various family arrivals and

departures, and the general madness of life in the early 2020s. In any case, although it has at times been exhausting, we're delighted to give this book its second outing!

Now get to it! `Book.play;`

I TUTORIALS

1 Beginner's Tutorial

David Michael Cottle

1.1 Hello World

This chapter will be your guide on an initial journey through the amazing, complex, esoteric, and sometimes confusing SC environment. It does not assume prior programming experience but does assume basic computer skills and some synthesis background; that is, you won't be surprised by terms such as voltage control, envelope, oscillator, or LFO. (If you would like to do some prep work, we will look at examples in subtractive synthesis, additive synthesis, phase modulation, and physical modeling.) You may also have some background in music and may be familiar with music software such as Logic Pro or Ableton Live. If you're an experienced programmer and trained musician, you might skim this material and move as quickly as possible to chapter 2.

The goal is getting interesting sounds as quickly as possible. To that end we'll often take liberties with the terms of object-oriented programming and the internal structure of SC, occasionally leaving out details so as to focus better on the topic at hand. Other chapters of this book will fill in the gaps.

This tutorial refers first and foremost to the SuperCollider IDE (integrated development environment; the dedicated cross-platform editor developed as part of the project) but all the material should be largely applicable to other editors. (See chapter 10 if you're curious.) Take a look at chapter 9 for information on downloading and configuring the IDE. You might also want to look through the list of Shortcuts in the preferences.

Finally, you may become a little lost during some of this discussion. Don't panic. This chapter is intended to push the envelope and set you up for later discussions. Just keep treading water, focus on the interesting sounds and the topics at hand, and your questions should be resolved further on.

Assuming you have downloaded and installed SC, launch it.

As the program starts, a workspace will appear with three "docklets"—an editor, a post window (which, as SC starts, will spout a barrage of cryptic lines), and a help browser. You can undock, close, or move these around. All these examples will be

entered into the editor, but keep the post window visible; it will print useful data as we move along.

In the untitled editor, type the line of code below, including quotes. (All examples of code intended to be typed and executed in SC will appear in the following font.)

```
"Hello world"
```

Place the cursor anywhere in that line and press [shift+enter] to evaluate this line. –> `Hello World` should appear in the post window. Congratulations! You've just written and executed your first SC program.

The example we just ran only involved the client side of SC, which essentially consists of the windows where you type and evaluate lines of code (i.e., commands for making sounds). These are sent to sclang, which is the application which interprets and executes SC code. Audio comes from the server side, a separate program that takes the client commands and turns them into sounds. We didn't need the server for the first example, which produced no sounds, but for the next one we do, so let's get it running. Type this line and evaluate (run) it.

```
s.makeWindow
```

A small window appears but also notice that the menus have changed. This is because SC switched you to the supercollider language, which in addition to actually evaluating the code is the place where most GUI (Graphical User Interface) elements can be found. In fact, the IDE (where you edit code), the language, and the server (which produces sound) are all separate programs which communicate with each other.

This server window can be used to control server behavior, though the IDE also provides options for this under the "Server" menu or by clicking on the bottom right of the main window. These latter approaches are more common, but a server window is sometimes useful, so it's good to know how to make one. This also allows us to demonstrate something important: Most of the windows SC produces (scopes, plots, open dialogs) will be presented from the language application. This can be a little disorienting; you run a line in the IDE, a window or dialog appears (switching to the lang app), you try to interact with the IDE again and nothing seems to work. Try switching back and forth so as not to be tripped up later in the chapter. I'll wait.

In the server window, click "Boot." (Also note the other controls; quit, "k" to kill, "m" to mute, volume slider, and a "record" button that you can engage at any time to save the audio to an AIFF file.) This sends a message to start the server app. As the server boots, the post window fills with useful information about what it sees on your system, the input and output devices. SC defaults to, and connects to, the currently

selected audio device for the system. This may result in an error about sample rates not matching. This often happens in the Mac OS, for instance, when you have separate input and output devices, or with Bluetooth headsets or buds where the signal going to the headsets needs to be 44.1k, but the mic is delivering lower rates, such as 22k. There are several fixes. Change your default audio device to wired headphones or computer speakers, or you can set the sample rate to what SC says it should be with, for example:

```
s.options.sampleRate = 48000;
```

You can also set input and output explicitly to any device in the list with this line. (Adjust to specify the desired device.)

```
s.options.device_("BlackHole 2ch");
```

Now delete everything (or open a new window) and type the following line, but don't evaluate it just yet.

```
play({SinOsc.ar(LFNoise0.kr(12!2, 600, 1000), 0.0, 0.1)})
```

You probably noticed the IDE has some editing aids. Auto-completion offers suggestions as it recognizes what you type. Press return to select (and arrows to highlight) the auto-complete suggestions. Even if you don't use return, this feature helps you type things correctly by showing a list of appropriate codes. Syntax is also colored by function, an indication it was typed correctly. For example, by default, "Hello World" was blue (a string), SinOsc should be light blue (a "Class"), numbers are purple, comments are gray, and so on. (You can customize these colors if you wish using the IDE's Preferences.) You should also see a yellow bar appear (in this example freq = 500.0, mul = 1.0, add = 0.0). These are prompts for arguments. They will make more sense as you do more examples. Finally, the most common typo when entering code is a missing parenthesis, bracket, or brace, resulting in a mismatched enclosure. If this happens, the segment that is not matched turns red if you place the cursor just outside the unmatched enclosure. (Try it!)

A few more precautions before we get sound: first, you should know how to stop playback. The menu Language/Stop stops all sounds, but most of the time you will find it's simpler to use its keyboard shortcut, so this is worth memorizing: Hold down the Cmd key on Mac and press the period (full-stop or "dot") key; Ctrl and the period key on most other platforms. Next, turn down your volume, warn others in the room, evict any domestic animals. When you're ready, select the line and choose any of the "evaluate" menu items or shortcuts.

Voilà! You should hear, as an appropriate introduction to SC, an early sci-fi computer sound effect.

Or, voilà, silence, and an error blurb in the post window, something like this:

```
ERROR: syntax error, unexpected SC_FLOAT, expecting ')'
in interpreted text
line 1 char 58:
play({SinOsc.ar(LFNoise0.kr(12!2, 600, 1000), 0.0 0.3)})
^^^
___
ERROR: Command line parse failed
```

Error messages are a common occurrence when developing code, so it's important to learn how to read them. In this case it's a parse error, which is usually just the result of a typo. The "^^^" indicates where, in the line above it, the program stopped understanding (couldn't parse). The mistake is usually somewhere just before that. Check for a missing comma (in this example, the comma before 0.3 is missing), a brace instead of a bracket, or incorrect case; any of them would cause an error.

Proofread, change, and try it again until you get the correct result. If you didn't get an error, try introducing one so you can get some practice reading them.

In this next example, almost a sort of simple physical model (similar to rubbing a fingernail along a guitar string), we'll experiment with some parameters:

```
play({RLPF.ar(Dust.ar([12, 15]), LFNoise1.ar(1/[3, 4], 1500, 1600),
0.02)})
```

Stop the sound and try replacing the numbers 12 and 15 with values between 1 and 100. Change the 3 and/or 4 to numbers between 1 and 10 and start it again. See if you can discern what aspect of the sound each number is controlling. (Hint: This example is *subtractive synthesis*, a resonant low-pass filter applied to low-frequency random impulses.) You can stop each sound between trials or pile them one on top of the other by evaluating multiple times.

Figure 1.1 shows an example spread out and indented into several lines, so be sure to select everything before evaluating. You can use Edit/Select All if it's the only thing in the file. It takes about 30 seconds to hear the full effect of this example.

```
// Listen for at least 30 seconds
(
play({
   var sines = 5, speed = 6;
```

```
    Mix.fill(sines,
         {arg x;
              Pan2.ar(
                   SinOsc.ar(x+1*100,
                        mul: max(0,
                             LFNoise1.kr(speed) +
                             Line.kr(1, −1, 30)
                        )
), rand2(1.0))})/sines})
)
```

Figure 1.1
Example of additive synthesis.

After the sound fades, stop the patch and try replacing 5 with 10, then reevaluate. Try values between 1 and 40. This value determines the number of sines added together. Stop again and change the speed (6) to values between 1 and 40, stopping and running between each value. Replace `LFNoise1` with `LFNoise0`. Swap the +1 and −1 to reverse the fade. Put everything except the −1 and +1 back to its original state (use undo), then replace `x+1*100` with `exprand(100, 10000)`. Next—and this will start to demonstrate some of SC's power—increase the number of sine waves (where it says `sines = 5`) from 5 to 10, 20, 40, 100. Finally, mix and match any of these changes.

Although it sounds like filtering, figure 1.1 illustrates *additive synthesis* (in which individual sine waves are summed) in a way that is audibly apparent in the actual sound—rich, complex, and yet you hear the individual frequencies emerge as the whole sound fades in or out. It shows how this method can be accomplished with just a dozen lines of code in SC. And, finally, it shows how quickly, with strategic changes, you can layer sounds that evolve from a few undulating tones to something very complex.

If these examples look like Greek, it's because SC is, to you, a foreign language that you're going to have to learn. If you're patient, you will soon be able to read and write SC code as you would any language. Look again at the additive synthesis example and see if you can recognize a few of the words with the help of this translation: This example tells the server to play a sound that is filled with a *Mix* of 5 *Panned Sine* Oscillators tuned to multiples of *100*, which are flashed on and off as a *Low-Frequency Noise* source pops up above a *maximum* of itself and 0, all slowly following a *Line* of attenuation over 30 seconds.

1.2 Messages and Arguments

A message is a lowercase word followed by a pair of parentheses containing a list of arguments. Arguments are separated by commas, inside parentheses, next to a message.

Together they look something like this: `message(arg1, arg2, arg3)`. Sometimes messages appear on their own, such as `SinOsc.ar(arglist)` or `Mix.fill(arglist))`, and sometimes they are connected to an uppercase word (discussed later) with a dot (for example `.ar` or `.fill` in constructions such as `SinOsc.ar(arglist)` and `Mix.fill(arglist))`. Scan the examples given so far and identify messages and argument lists. Note that each argument list is enclosed in parentheses.

Messages are commands, similar to commands you might give a person, such as jump, bake, or sing. Arguments are qualifications to these commands: jump 3 feet to the right, bake a pizza at 500˚ for 30 minutes, sing "Daisy, Daisy, Give Me Your Answer, Do." These commands, if they were available in SC, might look like this:

```
jump(3, right)
bake(pizza, 500, 30)
sing("Daisy, Daisy, Give Me Your Answer, Do.")
```

Argument order is important. If these examples were actual code, and the documentation for bake was "`bake(item, temp, min)`," then writing `bake(pizza 30 500)` would provide incorrect information on temperature and how long to bake, and might not have the desired results.

SC understands a number of synthesis- and composition-related commands. Let's try a few starting with `rand`, which generates a random number and takes a single argument. The argument sets the upper limit of the random number selection. Type and evaluate the line below several times:

```
rand(100)
```

Check the post window. You should see integers between 0 and 99.

Replace 100 with 100.0 and note the difference. As a *general* rule, if you use a float (short for floating point, e.g., 1.298, 178.287, 0.3) as an argument, a float is returned. Integers (e.g., 4, −55, 7, −11982, 1235) will return integers. (Also note that in SC both positive and negative floats smaller than 1 must have a leading zero: 0.1, −0.125, 0.003, 0.75.)

Here is another type of random message with two arguments. Can you determine the difference between `rand` and `exprand` and what the two arguments indicate? You may have to run it quite a few times to see a pattern:

```
exprand(1.0, 100.0)
```

The `exprand` message returns values from an exponential range (i.e., biased toward lower numbers). Notice that about half the choices are between 1 and 10, and the others between 10 and 100. Exponential random choices are useful in music (for frequency), since successive musical intervals (perceived by humans to be equal) are in reality exponentially larger gaps. (A4 to A5 is 440 to 880, a difference of 880, but A5 to A6 is 880 to 1760, twice as far.)

To test a larger set of numbers, place the code in a function (curly brackets) and append `!20`, which means evaluate 20 times.

```
{exprand(1.0, 100.0).round(0.01)}!20
```

In case you thought computers did anything truly random, try this line repeatedly, which chooses 20 supposedly "random" numbers

```
thisThread.randSeed = 666; {rand(10.0).round(0.01)}!20;
```

This is called *seeding* the random number generator, and 666 is the random seed. Try a less satanic number (66, 109923, 7, etc.), or a more correct satanic number (616). When you use a given seed, even on different computers (with the same OS), you will get the same "random" results (deus ex machina).

Here are some more messages. Try them one line at a time:

```
dup("echo", 20)
round([3.141, 5.9265, 358.98], 0.01)
sort([23, 54, 678, 1, 21, 91, 34, 78])
round(dup({exprand(1, 10)}, 100), 0.1)
sort(round(dup({exprand(1, 10)}, 100), 0.1))
```

You might guess from the name of each message what it does, but can you guess the significance of each argument? (Change the arguments to see.)

The first message, `dup`, is a duplicator. Its first argument, `"echo,"` is duplicated the number of times specified by the second argument; `round` rounds the first argument (the numbers inside the brackets—an array) to the precision of the second argument; `sort` takes a single argument—a collection of numbers, also an array—and sorts them. The next example picks 100 random numbers and rounds them. The last does all that, and then sorts them. Notice that the `exprand` and `dup`, previously on their own, are inside the argument list for `round`, which is inside `sort`. A message can be used as an argument inside another message. This is called *nesting*.

1.3 Nesting

To further clarify the idea of nesting, consider a hypothetical example in which SC could make you lunch. To do so, you might use a serve message. The arguments might be salad, main course, and dessert. But just saying `serve(lettuce, fish, banana)` is too general. To be more precise you could clarify those arguments, replacing each with a nested message and argument.

```
serve(toss(lettuce, tomato, cheese), bake(fish, 400, 20), [$$]mix(b
anana, ice cream))
```

SC would then serve not just lettuce, fish, and banana, but a tossed salad with lettuce, tomato, and cheese, a baked fish, and a banana sundae. These inner commands can be further clarified by nesting a `message(arg)` for each ingredient: lettuce, tomato, cheese, and so on. Each internal message produces a result that is in turn used as an argument by the outer message, as in figure 1.2.

```
serve(
    toss(
        wash(lettuce, water, 10), dice(tomato, small),
        sprinkle(choose([blue, feta, gouda]))
    ),
    bake(catch(lagoon, hook, bamboo), 400, 20),
    mix(
        slice(peel(banana), 20),
        cook(mix(milk, sugar, starch), 200, 10)
    )
)
```

Figure 1.2
Nested commands for hypothetical lunch-producing code.

When the nesting has several levels, we can use new lines and indents for clarity, as in figure 1.1. Some messages and arguments are left on one line, and some are spread out with one argument per line—whichever is clearer. Each indent level should indicate a level of nesting.

In figure 1.2, the lunch program is now told to wash the lettuce in water for 10 minutes and to dice the tomato into small pieces before tossing them into the salad bowl and sprinkling them with cheese chosen from three options. You've also specified where to catch the fish and to bake it at 400˚ for 20 minutes before serving, and so on.

Figure 1.3 makes use of nesting. Execute it, and then test your understanding by answering the questions that follow.

1. What is the second argument for LFNoise1.ar?

2. What is the first argument for LFSaw.ar?
3. What is the third argument for LFNoise1.ar?
4. How many arguments are in midicps?
5. What is the third argument for SinOsc.ar?
6. What are the second and third arguments for CombN.ar?
7. What is the only argument for play?

```
(
play(
    {
         CombN.ar(
              SinOsc.ar(
                   midicps(
                        LFNoise1.ar(3, 24,
                             LFSaw.ar([5, 5.123], 0, 3, 80)
                        )
                   ),
                   0, 0.4),
               1, 0.3, 2)
    }
)
)
```

Figure 1.3
Futuristic (circa 1956) nested music.

See this chapter's notes for the answers.[1]

1.4 Receiver.message, Comments

In the sound examples we've seen above, the uppercase words attached to messages with a dot, such as SinOsc in `SinOsc.ar` and LFNoise1 in `LFNoise1.kr`, are UGens ("unit generators"), which produce a stream of numbers suitable for digital audio. In more general terms, however, they are receivers. Messages are what to do. Arguments are how to do it. A receiver performs the action indicated by a message with several (or no) arguments.

There are many types of receivers (UGens, numbers, strings). When linked to a message, they can generate output, either individual values or collections. When a UGen is messaged, it may set up a process on the synthesis server to generate audio number streams. The dot between receiver and message loosely means "do this." The argument list describes how the receiver should complete the command. So `LFNoise1.kr(10,`

`100)` means "make a low-frequency noise generator of type 1, which will produce, 10 times per second, a random number between negative 100 and 100." Numbers, functions, arrays, and strings (text inside quotes) can also be told what to do by way of a dot and message.

In the next examples, we'll slip in a new convention: comments. Text that falls between two slashes and the end of a line is ignored during evaluation. This allows short explanations inside code. Comments are sometimes on a separate line but often are placed at the end of a line. These comments are a translation of the object message pairs. The dot more or less translates into "do this":

```
[45, 13, 10, 498, 78].sort // collection; sort
"echo".dup(20) // echo, do this; repeat yourself 20 times
50.midicps // the number 50, do this; convert yourself into Hz
444.cpsmidi // 444; convert yourself into a midi number
100.rand // 100; pick a number between 0 and yourself
{100.rand}.dup(50) // random picking function; repeat 50 times
[1.001, 45.827, 187.18].round(0.1) // collection of items;
"say \"I've just picked up a fault in the AE35 unit\" ".unixCmd
 // plot to kill me (macOS only)
```

Earlier, we saw `dup("echo", 20)` and `rand(100)`, but here we use `"echo".dup(20)` and `100.rand`. There is no difference, in terms of results. `100.rand` is receiver notation. `rand(100)` is functional notation. The first argument of a message can be placed in front of the message, separated by a dot; `message(arg)` and `arg.message`, or `message(arg1, arg2)` and `arg1.message(arg2)` are precisely the same. Which you use when is a matter of style. Things that begin with uppercase letters (`Mix`, `SinOsc`, `Pan2`, `Array`) are nearly always written using receiver.messages. Numbers, arrays, and text can be written either way, depending on whichever is clearer in a given context. A common syntax with receiver notation is to string together a series of messages as an alternative to nesting, as shown below. The result of each receiver.message pair becomes the receiver for the next message. Note the subtle but important distinction between the periods in `1000.0.rand`. The first is a floating point, indicating that the number is a float (i.e., a number with a fractional component after the decimal point, rather than an integer). The second is a dot that sends the message `rand` to `1000.0`. Here is an example a string of messages:

```
1000.0 // a number
1000.0.rand // choose a number between 0 and 1000
1000.0.rand.round(0.01) // choose a number and round it
1000.0.rand.round(0.01).post // choose, round, then post
{1000.0.rand.round(0.01).postln}.dup(100).plot // choose, round,
```

```
 dup, plot
{1000.0.rand.round(0.01)}.dup(100).postln.sort.plot // choose   et
c., sort, plot
```

1.5 Enclosures

There are four types of enclosures in the examples above: (parentheses), [brackets], {braces}, and quotation marks ("some text"). It is important to keep these pairs matched, or "balanced," as you write code. Nesting can make balancing difficult and confusing. Later in the chapter we'll see other (often clearer) methods, such as assigning items to variables, but the SuperCollider language editor has some tools to help you keep track. As you type the closing enclosure, its match (and the code it encloses) is highlighted in gray. Selecting a bracket (or positioning the cursor right before or after it) has the same effect. The menu item Edit/Select Enclosing block repeatedly shows balanced enclosures. In fact, this command, along with Select Region, is a quick way to select large sections of code for evaluation, deletion, or copy/paste operations. Note that figure 1.3 is enclosed in parentheses, which in this case has no function other than defining a block for quick evaluation. This convention of code, to be executed as a block being wrapped in parentheses, is ubiquitous in SC code.

Quotation marks are used to enclose a string of characters (including spaces) as a single unit. These are aptly called strings. Single quotes create symbols, which are similar to strings. You can also make a symbol by preceding some text with a backslash. Thus `'aSymbol'` and `\aSymbol` are equivalent. (Note with the backslash you can't include spaces.) Symbols are often used as labels for parameters and are sometimes, but not always, interchangeable with strings. See the `Symbol` and `String` help files for more information.

Parentheses are used to enclose argument lists, but they can also force precedence. Precedence is the order in which math operations are performed. As an example, recall your math training and evaluate this expression: 5 + 10 * 4? Try this line to test it:

```
5 + 10 * 4
```

In normal mathematics it would be 45. (The rule is: multiply and divide then add.) In SC it's 60. That's because precedence with binary operators (+, −, *, /, etc.) in SC is left to right, regardless of operation. The equation above calculates 5 + 10, then that result is multiplied by 4. It's an easy rule to remember, but you can also force precedence with parentheses: (5 + 10) * 4 = 60, but 5 + (10 * 4) = 45.

Precedence can also be forced when combining binary operations and messages, where messages normally take precedence. For example, `5 + 10.squared` results in

105 (10 squared + 5), whereas `(5 + 10).squared` results in 5 + 10, which is then squared: 225. This is a very different result, and demonstrates a potential cause of errors.

The next enclosure, already used in several patches, is a bracket. Brackets define a collection of items. An `Array` (one type of collection) can contain numbers but also text, functions, or entire patches. You can even mix data types within an array. Arrays can receive messages such as `reverse`, `scramble`, `mirror`, `rotate`, `midicps`, `choose`, and `permute`, to name a few. You can also perform mathematical operations on arrays:

```
[0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5].reverse   // retrograde of a
12-tone row
12-[0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5].reverse   // retrograde i
nversion
([0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5] + 5) %12   // transposition
[0, 2, 4, 5, 6, 7, 9, 11].scramble // diatonic scale
[60, 62, 64, 67, 69].mirror // pentatonic
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].rotate
[60, 62, 64, 65, 67, 69, 71].midicps.round(0.1) // convert   midi t
o frequency in Hz
[1, 0.75, 0.5, 0.25, 0.125].choose // maybe durations?
0.125 * [1, 2, 3, 4, 5, 6, 7, 8].choose // multiples of   a smalles
t quantize value
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].permute(6)
```

More on arrays later.

Braces, the last enclosure, define functions. Functions perform specific tasks that are usually repeated, often millions of times and often with different results. Try these lines, one at a time:

```
exprand(1, 1000.0)
{exprand(1, 1000.0)}
```

The first line picks a random number, which is displayed in the post window. The second prints a very different result: a function. What does the function do? It picks a random number. How can that difference affect code? Consider the lines below.

```
dup(rand(1000.0), 5) // picks a number, duplicates it
dup({rand(1000.0)}, 5) //duplicates the function of picking   a num
ber
// essentially, this (which has a similar result)
```

```
[rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0),   rand(100
0.0)]
```

The first line picks a random number and duplicates that random number. The second duplicates the function that picks random numbers and evaluates that, which produces 5 *different* random numbers. Functions understand a number of messages: plot, play, scope, and dup, to list a few. (Note that the plot and scope messages switch you to the SuperCollider language app as noted above. You will have to return to the IDE to continue running the lines of code.) Turn the speakers down.

```
{LFNoise0.ar(10000)}.plot // plot the output of this UGen
{LFNoise0.ar * 0.1!2}.play //play noise, basically a series   of ra
ndom numbers
{LFNoise0.ar(10000!2)*0.4}.scope // play and show on a scope
{100.rand}.dup(10) // pick 10 random numbers
{100.rand}! 10 // same as above
{100.rand}.dup(10).postln.plot // pick 10 numbers, post, then   plo
t them
{100.rand}.dup(100).sort.plot // pick 100 numbers, sort them,   the
n plot
```

Take a moment to scan all the examples in this chapter, maybe even venture into other chapters, and see if you can identify these key elements of code:

Receivers (objects), messages, and argument lists—Object.message(arglist)

Collections—[list of items]
Functions–{often multiple lines of code}
Strings—"words inside quotes"
Symbols—'symbols' or \symbols

For a more detailed discussion of SuperCollider's syntax, see chapter 5.

1.6 Multichannel Expansion

Arrays have many applications, but multichannel expansion is one that borders on magic. If an array is used as any argument in a UGen, the entire patch is duplicated. The first copy is sent to channel one, using the first value of the array in that patch; the second copy goes to channel two with the second value of the array. Note that the only difference between the lines below is the array for the first argument of the `LFNoise.kr`

```
{Blip.ar(50, LFNoise0.kr(5, 12, 14), 0.3)}.play // single   channel
{Blip.ar(100, LFNoise0.kr([5, 10], 12, 14), 0.3)}.play   // stereo
{Blip.ar(30, LFNoise0.kr([5, 10, 2, 25], 12, 14), 0.3)}.play   // q
uad
{Blip.ar(25, LFNoise0.kr([5, 4, 7, 9, 5, 1, 9, 2], 12, 14),   0.
3)}.play // 8 channel
```

The first example is in mono, output to the left speaker (assuming a stereo setup), generating random filter cutoffs at a rate of 5 times per second. The next is stereo with a rate of 5 times a second in the left channel while the right is playing at 10. The next is (yes, it's that easy) quad; 5 in front left, 10 in front right, 2 in back left (depending on your setup), 25 in back right. If your interface supports only two channels, as most do, the last two examples will run, but you won't hear anything beyond the second incarnation. The last line produces eight discrete audio signals, each with a different LFO rate.

If the other arguments are arrays, the values are matched up accordingly. For example, if one argument has [45, 32, 66, 19], and another has [5.3, 35.1], then the second array will be duplicated ([5.3, 35.1, 5.3, 35.1]); channel 1 will use 45 and 5.3, channel 2 will use 32 and 35.1, channel 3 will use 66 and 5.3, and channel 4 will use 19 and 35.1.

Note that if you create a patch with no stereo expansion, it will play mono, and only in the left speaker or first output. An exclamation point duplicates an item into an array. So 32!2 will become [32, 32], and it can be used as a quick way to make a patch mono, but in both speakers.

Multichannel expansion illustrates one of the ways in which SC is both powerful and economical. Simply adding an array gives you multiple channels, whereas synths with graphical interfaces, modules, and patching, while possibly and only initially more intuitive, are severely limited by the interface. You would have to duplicate all the objects and patch cords one at a time to reproduce a 5.1 patch. But in SC, it's just a few lines of code. Duplicate that for 4 speakers? Type 6 extra characters and press Ctrl/Cmd+enter. Take *that*, uh, well, pretty much all graphic interfaces.

Naturally, to make the most of SC it is important to know what objects are available, what messages they understand, what argument lists accompany the messages, and what kind of output they produce.

1.7 Help!

Items with capital letters (`SinOsc`, `LFSaw`, `LFNoise`, `PMOsc`, `Array`, `Mix`) and many messages (`midicps`, `max`, `loop`, `randomSeed`) have Help files. Select any of these items, or simply place the cursor anywhere in the item and ask for help (Help/Look up

documentation for cursor). A Help file appears with a description, a list of arguments, and examples.

Test this with a new UGen: `PMOsc`. Type that word into any window, leave the cursor at the end open, and read the Help file. Just skip over the terms you don't know. Most important, try the examples.

`PMOsc` is a phase modulation oscillator, the more efficient cousin of frequency modulation. The gist of amplitude, frequency, and phase modulation is achieving rich harmonic spectra (i.e., an interesting sound) at very little cost (CPU). In the days of modular synthesis, even top electronic studios had, at best, a dozen oscillators for experiments. Frequency modulation (FM) synthesis required only two, so it was an effective procedure back then. Even today phase modulation can be an efficient path to complex sounds. This sound has an edge. As usual, turn down the speakers.

```
{PMOsc.ar(440, 550, 7)}.play // like an FM radio, which also   uses
'FM'
```

The Help file reveals that arguments for `PMOsc.ar` are carrier wave, modulator, index, modulator phase, and then `mul` and `add`. Experiment with the arguments by stopping, changing, and running the patch again. Try values between 100 and 1000 for arguments one (carrier) and two (modulator), and values between 1 and 8 for the third (index). Make it stereo. In this example, we don't supply arguments for `modphase`, `mul`, or `add`. In most cases, UGen will use defaults if no values are given.

Now let's make this example more interesting by nesting. As we've seen, an argument can be a single static number (e.g., 440, 550, 7) or a complete nested UGen, such as `LFNoise1`, `Line`, or, better yet, `MouseX` or `MouseY`. These are mouse controls that understand the `kr` message with `minval`, `maxval`, `warp`, and `lag` as arguments (read each Help file for details). `MouseX` and `MouseY` generate a continuous output between low and high arguments. In the example below, their output is being used as the second and third arguments in PMOsc, allowing you to explore how each argument in these ranges affects the sound:

```
{PMOsc.ar(440, MouseY.kr(1, 550), MouseX.kr(1, 15!2))}.play
```

Arguments have to be in the correct order to work properly. If you mix them up or skip one, bad (or, occasionally, unexpected and exciting) things happen. There are situations, however, where it is convenient to skip or reorder arguments. In these cases, we use keyword assignment. If you precede an argument with its keyword (listed in the Help file) and a colon, you identify that argument explicitly, regardless of its position in

the argument list. To illustrate, all of the following lines have precisely the same meaning and result:

```
{PMOsc.ar(100, 500, 10, 0, 0.5)}.play // all arguments listed   in
 order
{PMOsc.ar(carfreq: 100, modfreq: 500, pmindex: 10, mul: 0.5)}.   pl
ay // keywords
{PMOsc.ar(100, mul: 0.5, pmindex: 10, modfreq: 500)}.play   // mixe
d
```

This has a number of advantages. First, it's a type of documentation similar to comments, identifying each argument by name. Second, it can make the argument list more portable. When experimenting, you may want to swap out UGens. The argument lists for each UGen may be interchangeable, but usually they are not. For example, in `Blip.ar(400, 10, 0.5)`, 400 is the frequency, 10 is the number of harmonics, and 0.5 is the `mul`. But if we used that same argument list in `LFNoise1` (i.e, change the Blip to LFNoise1 but keep the same arguments; `LFNoise1.ar(400, 10, 0.5)`), the arguments have different meanings; 400 is frequency, but 10 is now `mul`, and 0.5 is the add: a very different result. (Don't try it.) But using keywords—`Blip.ar(freq: 400, mul: 0.5)`, `SinOsc.ar(freq: 400, mul: 05)`, and `LFNoise1.ar(freq: 400, mul: 0.5)`—may have more similar results because the arguments are explicitly identified. The worst that could happen is a warning if the UGen has no such keyword.

Finally, we can list arguments in any order, or even leave them off (to illustrate `mul` in a simple `SinOsc`, for example) without all that typing. Note that if you mix ordered and keyword arguments, the ordered ones *must* come first.

```
{SinOsc.ar(mul: MouseX.kr(0, 1.0))}.scope
```

The argument for `mul` can do some very clever things, depending, for instance, on what its UGen is plugged into, but in this example, it controls the volume of the sound that we hear.

Using the `MouseX` in these examples illustrates how parameters such as amplitude, carrier frequency, modulation index, and phase can be manually controlled. You can also build patches in which parameters are modified using an automated process, for example, a `Line.kr`. (This is technically different from, but analogous to, voltage control.) Look up the Line Help file, note its arguments, and try the examples. Then, based on your experiments with `PMOsc` and the mouse controls, modify these examples using three Line.kr UGens inside a `PMOsc`, one for each of the first three arguments in the PMOsc. One solution appears in the notes section.[2] Remember, this very complex sound is essentially just 2 oscillators.

Use the code in figure 1.4 to practice getting help. It is modeled after a classic voltage-controlled oscillator (VCO), a voltage-controlled filter (VCF; although in Blip, it's not *technically* a filter), and voltage-controlled amplifier (VCA). Use multichannel expansion (or add `!2` after any number) to make it stereo.

```
(
{
Blip.ar(
        TRand.kr (// frequency or VCO
              100, 1000, // range
              Impulse.kr(Line.kr(1, 20, 60))), // trigger
        TRand.kr (// number of harmonics or VCF
              1, 10, // range
              Impulse.kr(Line.kr(1, 20, 60))), // trigger
        Linen.kr (// mul, or amplitude, VCA
              Impulse.kr(Line.kr(1, 20, 60)), // trigger
              0, // attack
              0.5, // sustain level
              1/Line.kr(1, 20, 60)) // trigger
        )
}.play
)
```

Figure 1.4
VCO, VCF, VCA.

Here's this example explained from the inside out: three instances of Line.kr send a number to the three `Impulse` UGens, which send triggers to both `Linen` and `TRand`. At each trigger, the two `TRand` UGens generate values between 100 and 1000 and between 1 and 10, the first being used for the `freq` argument in Blip (frequency), and the other for the second argument of `Blip`, `numharm` (number of harmonics).

You might have noticed that we've used two messages for UGens that generate signal: `.kr` and `.ar`. Both the `kr` and `ar` messages tell a UGen to send a stream of numbers produced in a given fashion: a sine wave for `SinOsc`, a triangle wave for `LFTri`, or a saw wave, or a pulse, or a noise (something random) in the case of `TRand`. The Blip generates sounds we actually hear, so the stream of numbers needs to be produced at an audio rate (e.g., 44.1 kHz, depending on your sound card settings). The `TRand`, `Impulse`, and `Line` are controls and need to supply only a dozen or so values per second, so their resolution doesn't need to be as fine. The `ar` message generates values at an audio rate (e.g., 44.1k, 48k), and `kr` generates them at a control rate (depending on your system, but usually less than 1k). Using `kr` requires fewer calculations and saves on processing.

Another `Impulse` sends a trigger to the `Linen`, which is an envelope. It shapes the `mul` (amplitude) of each event. The arguments for `Linen` are trigger (or gate), attack time, sustain level, and decay. At each trigger (the rate of which is controlled by a `Line`) the `mul` argument of the `Blip` will move from 0 amplitude to a held level of 0.5 (out of a maximum of 1) in 0 seconds (attack time), then decay back to 0 in `1/Line` seconds (decay time).

Why `1/Line` seconds? The `Impulse` is sending a series of triggers to `Linen`. The rate of those triggers is controlled by `Line`, which begins at 1, then increases to 20 times per second over a period of 60 seconds. The duration between each event is the reciprocal of the trigger rate. When the trigger is 20, the time period between each trigger is 1/20 of a second. It would make sense, then, for the duration of each envelope to be the same as the time between each trigger. Since the attack time is 0, we can just focus on the decay, so decay time = duration. Using `1/Line` links the two so that duration will always be the reciprocal of trigger rate. As the triggers change, so does the duration of the envelope for each event (e.g., 5 per second, each one a fifth of a second in duration, 2 times per second, each one half a second long; that is, when Line = 2, 1/Line = 1/2).

Notice the redundancy in this code. We've created nine UGens where three would do. Since the three Impulses, two TRands, and four Line UGens are identical, wouldn't it be easier to use one of each, somehow linking them all?

Yes, and to do that we use variables.

1.8 Variables

The default set of SC variables is just what you remember from algebra: lowercase letters from *a* through *z* that were used in expressions like $x * y = 60$, and if $x = 10$, y must be 6.

You can store numbers, words, UGens, functions, or entire patches in any of these variables, using the equal (=) sign (although by convention a few are reserved; for example, the variable *s* is always used for the server). That variable then can be spread around the patch, linking important functions. Here is a simple math example:

```
(
a = 440;
b = 3;
c = "math operations";
[c, a, b, a*b, a + b, a.pow(b), a.mod(b)]
)
// more verbose, but exactly the same result
["math operations", 440, 3, 440*3, 440 + 3, 440.pow(3),   440.mod
(3)]
```

These examples are a collection of statements (lines of code ending with a semicolon). Pages and pages of code are usually broken up into a series of statements. They are executed in order, such as saying "do this; then this; now do this; and finally, this (and return the result)." The order of the steps is important because we have to give variables a value before they are used.

If the last two examples have the same result, what is the point? The first reason is efficiency. We can make one change to the variable *b*, and that change is carried over to all the subsequent operations. The second reason is to link-related functions, for example, a trigger and an envelope duration.

Here is an example using the variable *r* to hold a MouseX UGen:

```
(
{
r = MouseX.kr(1/3, 10);
SinOsc.ar(mul: Linen.kr(Impulse.kr(r), 0, 1, 1/r))
}.play
)
```

Using variable *r*, we link the trigger rate for `Impulse` to the duration of each envelope. When *r* is 1/3 (one every 3 seconds), the duration of the envelope is 3 seconds (and if *r* is 1/3, then 1/*r* is 3). When *r* is 10, then the decay rate is 1/10. To illustrate the difference, change 1/*r* to a static number (e.g., 1).

Figure 1.5 shows a more complex synthesis example with variables and statements. Notice the commented alternatives. When you remove the comment marks from in front of a line, say, `r = LFTri.kr(1/10)*3 + 7`, that line replaces the `r = Line.kr(1, 20, 60)` above it, swapping the line control for a low-frequency triangle wave. For clarity and efficiency, you should probably also "comment out" the first line, but in short examples or demonstrations, it's not necessary.

```
(
// run this first
p = {// make p equal to this function
    r = Line.kr(1, 20, 60); // rate
    // r = LFTri.kr(1/10) * 3 + 7;
    t = Impulse.kr(r); // trigger
    // t = Dust.kr(r);
    e = Linen.kr(t, 0, 0.5, 1/r); // envelope uses r and t
    f = TRand.kr(1, 10, t); // triggered random also uses t
    // f = e + 1 * 4;
    Blip.ar(f*100, f, e) // f, and e used in Blip
```

```
}.play
)

p.free // run this to stop it
```

Figure 1.5
Synthesis example with variables and statements.

This version is more efficient because the variable *t*—a single `Impulse`—operates in three different places and the variable *r*—a single Line—is used in two places. The `TRand`, stored in *f*, is generating values between 1 and 10. It is used by itself as the `numHarm` argument, and when multiplied or scaled by 100, it will work for frequency, too: 100 to 1000. (See the discussion on scale and offset below.) The first version of this patch uses 10 UGens. The second, with variables, uses only four, a 60 percent reduction. That is significant.

In addition to efficiency, we have linked all three triggers, and hence the duration of each event, together. This is possible only when using a common variable.

There are many situations in which you want to link one parameter to another. Phase modulation is one. In PM, two frequencies are combined as the modulator and carrier. Sidebands emerge from the blended waves and contribute to the quality of the sound. We can provide independent values or controls for each, and that's interesting, but if we describe the modulator as a function of the carrier, the composition of sidebands will remain constant (relative to the carrier) when the carrier changes. This ensures a consistent timbre, and our ears track the fundamental frequency better. In other words, if the modulator is expressed as a ratio of the carrier, we hear pitch changes with the same timbre. In the second example in figure 1.6, MouseX controls the modulator ratio. Use the mouse to fish for an interesting timbre, but note that once you settle on one position, the timbre is consistent, even though the carrier frequency changes.

There is one problem with the default variables `a` through `z`. As single letters, it's hard to know what their function is. For this reason, one should use them only for short examples and tests. For larger patches, you should *declare* your own variables with names that better reflect their functions in the patch. Variable names must begin with the lowercase letters *a* through *z* (you can use numbers or underscores within the name, just not as the first character), and they must be contiguous (no spaces or punctuation). Note that you can assign a value to a variable at declaration (`var rate = 4`).

Time for a practical test. Beginning with figure 1.6,

1. Make it stereo.
2. Add a Line.kr to move the index (now 12) from 1 to 12.
3. Identify the arguments for each receiver.message pair.

4. Control the rate, using another Line with a range of 1 to 20.
5. Add keywords to all argument lists.
6. Add a new variable called `env`, and assign it to Linen (an envelope) with an attack of 1/rate and a decay of 0 to control the `mul` of the `PMOsc` (use keyword assignment).
7. Figure out what the * 500 + 700 are doing. A hint: the `LFNoise0` is a bipolar UGen, meaning that it generates values between −1 and +1.

```
(
{// carrier and modulator not linked
   r = Impulse.kr(10);
        c = TRand.kr(100, 5000, r);
        m = TRand.kr(100, 500, r);
        PMOsc.ar(c, m, 12)*0.3
   }.play
)
(
{
   var rate = 4, carrier, modRatio; // declare variables
   carrier = LFNoise0.kr(rate) * 500 + 700;
   modRatio = MouseX.kr(1, 2.0);
   // modulator expressed as ratio, therefore timbre
   PMOsc.ar(carrier, carrier*modRatio, 12)*0.3
}.play
)
```

Figure 1.6
Phase modulation with modulator as ratio.

See the notes section for one version.[3]

1.9 Synth Definitions

When running each example, you probably noticed a line in the post window with the text `Synth("temp_number": 1000)`. Each set of interconnected UGens is packaged into a `SynthDef` (i.e., a Synth definition), which describes which UGens are used and how they are plugged together. The server can then use this definition to make running Synths based on that synthesis recipe. When you use `{<code>}.play`, SuperCollider does the work for you under the surface, such as auto-naming the associated SynthDef and using it straight away to create a running Synth, with the assigned temporary name. We can be more explicit about this process, specifying a name by wrapping a patch in a

`SynthDef`. In addition to the name, we have to explicitly identify the output bus (covered next) with the `Out.ar` message:

```
{SinOsc.ar}.play // generates a temp__number synth
// names the SynthDef and output bus 0 (left) explicitly
SynthDef("sine", {Out.ar(0, SinOsc.ar)}).play // create   a synthde
f and play
SynthDef("sine", {Out.ar(1, SinOsc.ar)}).play // right channel
// or
(
SynthDef("one_tone_only", {
   var out, freq = 440;
   out = SinOsc.ar(freq);
   Out.ar(0, out)
}).add // make sure SuperCollider knows about this SynthDef
)
// then use it to create a running Synth
Synth("one_tone_only");
```

Note here two ways of invoking the SynthDef, the first being to `play` it, just like the `{}.play` construction, wrapping up the definition and use of the definition in one go. The second is to add the SynthDef, adding it to the list of SynthDefs that SuperCollider's synthesizer knows about. We then explicitly and separately create a Synth that uses the new SynthDef. In practice, there are a variety of messages for setting up a SynthDef, to pass itself over to the server (`send(s)`), or to write itself to the hard drive ready for future reuse (`writeDefFile`), among others. You may read more about these variants in the `SynthDef` help file if you're curious, but `add` and `play` will do for now; `add` in particular will be the primary mechanism for creating SynthDefs in this book.

We named this SynthDef "one_tone_only" because the variable for frequency cannot be changed once the SynthDef is established. It will always play 440, not very useful outside of tuning. It is possible to change the parameters of Synths once they are playing (although not their chain of UGens), but to implement that important flexibility, we must use arguments instead of variables for the SynthDef. Arguments create Controls that allow you to pass different values to a Synth while it is running. They are declared at the start of the function within the SynthDef and should be assigned defaults.

```
SynthDef("different_tones", {
   arg freq = 440; // declare an argument and give it   a default v
alue
   var out;
```

```
    out = SinOsc.ar(freq)*0.3;
    Out.ar(0, out)
}).play
```

These arguments are similar to the ones we've been using to set and change parameters on UGen message pairs, but the syntax for calling them is a little different. The first argument for Synth is the name of the SynthDef; the second is an array with pairs of values matching the synth argument name with the value you want to use, `["arg1", 10, "arg2", 111]`. The name can be identified with `freq` or `\freq`. (In this case, symbols and strings *are* interchangeable.)

```
// Run all four, then stop all
Synth("different_tones", ["freq", 550]);
Synth("different_tones", [\freq, 660]); // same as "freq"
Synth("different_tones", ["freq", 880]);
// If no argument is specified, defaults are used (440)
Synth("different_tones")
```

In the previous example, we had to stop all synths at once, using Lang/Stop (a sort of sledgehammer method). But a synth combined with a variable assignment allows several instances to be tracked and controlled independently, using commands to free (stop) or set (change) an argument in flight. Run all of these lines one at a time.

```
a = Synth("different_tones", ["freq", 64.midicps]);
b = Synth("different_tones", ["freq", 67.midicps]);
c = Synth("different_tones", ["freq", 72.midicps]);
a.set("freq", 65.midicps);
c.set("freq", 71.midicps);
a.set("freq", 64.midicps); c.set("freq", 72.midicps);
a.free;
b.free;
c.free;
```

Figure 1.7 shows a `PMOsc`, configured to produce a crotale-like sound, set inside a definition with arguments. This one has a few new ideas (such as `Env` and `doneAction`) that are not germane to our discussion. You can read about them in the Help files or other chapters. Sending the synth definition to the server will not result in any sound. It adds it to the list of synth types that the server can then make instances of, akin to calling up a preset on a synthesizer. Executing the last line multiple times creates and plays synths based on that definition, sending different values for frequency and index each time.

```
(
// run this first
SynthDef("PMCrotale", {
   arg midi = 60, tone = 3, art = 1, amp = 0.8, pan = 0;
   var env, out, mod, freq;

   freq = midi.midicps;
   env = Env.perc(0, art);
   mod = 5 + (1/IRand(2, 6));

   out = PMOsc.ar(freq, mod*freq,
        pmindex: EnvGen.kr(env, timeScale: art, levelScale: tone),
        mul: EnvGen.kr(env, timeScale: art, levelScale: 0.3));

   out = Pan2.ar(out, pan);

   out = out * EnvGen.kr(env, timeScale: 1.3*art,
        levelScale: Rand(0.1, 0.5), doneAction:2);
   Out.ar(0, out); // Out.ar(bus, out);
}).add;
)

// Then run this a bunch of times:

Synth("PMCrotale", ["midi", rrand(48, 72).round(1), "tone",   rrand
(1, 6)])
```

Figure 1.7
Synth definition.

Unless explicitly removed or replaced, this definition will stay with the server until it quits, and we can access it anytime. We will use it in some examples later, so if you come back to this after restarting SC, you will need to execute the SynthDef code again.

1.10 Buses, Buffers, and Nodes

Buses are used for routing audio or control signals. There are 1,024 audio buses and 16,384 control buses by default, but you can change this if you like (see the ServerOptions Help file). The audio outputs and inputs of your sound hardware are numbered among the audio buses, coming at the start of the numbering with outputs first (recall `Out.ar(0)`; i.e., sending to the first audio bus, equivalent to your first output), then inputs (by default beginning with bus 8, though this will depend on the number of

output channels set in ServerOptions). The other buses are "private" and are for internal routing purposes such as parallel controls and fx, where you route one control source (such as a random wave) to several sounds and/or several sounds to a single effect (such as a delay). There are two advantages to parallel controls and effects: they are more efficient (similar to the variables example), and a single change to a control or effect will change all sounds to which it is connected.

As an example of parallel routing, imagine you have a five-member band and want to improve their sound with some reverb. You could buy a reverb unit for each musician, but wouldn't it be more efficient to route them all to a mixer that is connected to one reverb? That is less expensive, and if you want a longer delay time for everyone, you change that parameter on only one unit. How do you connect everyone to a single control or effect in SC? With a bus.

For these experiments let's try a new UGen: `PlayBuf`. It plays an audio file, allowing complex, real-time concrete treatment such as looping and speed change. First, we have to read an audio file into a buffer. Well, first, we need two audio files to experiment with. You can open any audio file, but for compatibility, they should be 44.1k, mono, and at least five seconds. You can use an audio editor such as Audacity or Amadeus Pro to create a short clip. For the actual sound, maybe a sound effect, Hendrix, Webern (what I used at this writing), or anything by Tom Waits. Of course, in the spirit of musique concrète, you can record your own audio. We'll assign the buffers to the *environment variables* ~b1 and ~b2. Environment variables are declared using the tilde (~) character and are similar to *a* through *z*, but user defined. They will work anywhere in the patch, in other patches, or even in other windows. By contrast, the variables in figure 1.7 (`env`, `out`, `mod`, and `freq`) will work only inside the function where they are declared. (For more on variable scope and environments, see chapters 5 and 8.) We need environment variables for these examples because we'll be using them in lots of places, including examples later in this chapter. If SC crashes or you are interrupted, run these lines again before continuing:

```
~b1 = Buffer.loadDialog(s);
~b2 = Buffer.loadDialog(s);
{PlayBuf.ar(1, ~b1)}.play; // number of channels and buffer.
{PlayBuf.ar(1, ~b2)}.play; // number of channels and buffer.
[~b1.bufnum, ~b1.numChannels, ~b1.path, ~b1.numFrames]
[~b2.bufnum, ~b2.numChannels, ~b2.path, ~b2.numFrames]
```

The last two lines show information for each buffer, such as how many frames it has, and therefore how long it is (divide the frames by the channels and sample rate for duration) (figure 1.8). `PlayBuf` can be looped using its `loop` argument (1 = loop, 0 = don't loop), but here a trigger is used to reset the playback, looping only a section of the

file (see also `TGrains`). The trigger for the right channel is slightly later than the left one (by 0.01), allowing the loops to slowly shift and "come out" of phase (as in Steve Reich's music). The position of the loop is also gradually increased with the `Line.kr`. The envelope is used to chop off the first and last hundredths of a second for a cleaner transition.

```
~b1 = Buffer.loadDialog(s); // rerun this if you need to
(// phase shift
{
   var rate, trigger, frames;
   frames = ~b1.numFrames;
   rate = [1, 1.01];
   trigger = Impulse.kr(rate);
   PlayBuf.ar(1, ~b1, 1, trigger, frames * Line.kr(0, 1, 60)) *
   EnvGen.kr(Env.linen(0.01, 0.96, 0.01), trigger) * rate;
}.play;
)

(// speed and direction change
{
   var speed, direction;
   speed = LFNoise0.kr(12!2) * 0.2 + 1;
   direction = LFClipNoise.kr(1/3);
   PlayBuf.ar(1, ~b1, (speed*direction), loop: 1);
}.play;
)
```

Figure 1.8
Playback buffers.

The second example adds controls for playback speed and direction.

You have two sounds loaded into buffers. What if you want to use one control on both PlayBufs? Similarly, what if you would like to switch between several controls (`LFNoise1`, `LFNoise0`, and `SinOsc`) with either PlayBuf, applying them independently? What if you want to apply two controls at the same time?

Building separate synths for each of those configurations is possible, but it is inefficient and limits you to those designs. Better to create the sources and controls as modules (like vintage modular synths) and connect them using buses (like virtual patch cords). For control rate signals, we will use control buses. To connect them we use `Out.kr` and `In.kr`. The arguments for `Out` and `In` are the bus number (anything up to

4096) and the number of channels. To connect an output to an input, you would use the same bus number:

```
Out.kr(1950, SomeControl.kr), then SinOsc.ar(In.kr(1950, 2));
```

The usual way of doing this is to use a Bus object, as shown in figure 1.9, which will supply an index for you. If you work this way, you can pass the variable it is assigned to directly, without needing to access its index. It also makes your code more reusable since you won't need to worry about index conflicts. You won't hear any sound in the first example because it has no control (nothing is connected to the `In.kr` buses).

```
~b1 = Buffer.loadDialog(s); // rerun this if you need to
~b2 = Buffer.loadDialog(s); // rerun this if you need to
(// you won't hear any sound yet
~kbus1 = Bus.control; // a control bus
~kbus2 = Bus.control; // a control bus
{
   var speed, direction;
   speed = In.kr(~kbus1, 1) * 0.2 + 1;
   direction = In.kr(~kbus2);
   PlayBuf.ar(1, ~b1, (speed * direction), loop: 1);
}.play;
)
(
// now start the controls
{Out.kr(~kbus1, LFNoise0.kr(12))}.play;
{Out.kr(~kbus2, LFClipNoise.kr(1/4))}.play;
)

// Now start the second buffer with the same control input busses,
// but send it to the right channel using Out.ar(1 etc.

(
{
   var speed, direction;
   speed = In.kr(~kbus1, 1) * 0.2 + 1;
   direction = In.kr(~kbus2);
   Out.ar(1, PlayBuf.ar(1, ~b2, (speed * direction), loop: 1));
}.play;
)
```

Figure 1.9
Connecting controls with a bus.

We started the two control sources after the first PlayBuf so you could hear there was no sound until they were running. In the next example we start the controls first and show them on scopes. When the scopes appear, they are the frontmost window in the SC language, so be sure to switch back to the SC code window before running the next example. Stop playback and run the last two lines again on their own to see what it would sound like without the controls.

```
~kbus3 = Bus.control; // a control bus
~kbus4 = Bus.control; // a control bus
// run these one at a time, (turn down the speakers!)
{Out.kr(~kbus3, SinOsc.kr(3).scope("out3") * 100)}.play;
{Out.kr(~kbus4, LFPulse.kr(1/3).scope("out4") * 200)}.play;
{Out.ar(0, SinOsc.ar(In.kr(~kbus3) + In.kr(~kbus4) + 440).   scope
("left"))}.play;
{Out.ar(1, SinOsc.ar(In.kr(~kbus3) + In.kr(~kbus4) + 880).   scope
("right"))}.play;
```

When combined with a SynthDef, the routing can be swapped around in real time.

```
~kbus3 = Bus.control; // a control bus
~kbus4 = Bus.control; // a control bus
{Out.kr(~kbus3, SinOsc.kr(3).range(340, 540))}.play;
{Out.kr(~kbus4, LFPulse.kr(6).range(240, 640))}.play;
SynthDef("Switch", {arg freq = 440; Out.ar(0, SinOsc.ar(freq, 0, 0.
3))}) .add
x = Synth("Switch"); // default
x.map(\freq, ~kbus3)
x.map(\freq, ~kbus4)
```

Similarly, an audio signal can be routed through a chain of effects using audio buses. To illustrate, figure 1.10 shows two effects for our PlayBufs: a modulator and a delay. (The two buffers, ~b1 and ~b2, should still contain the sounds we loaded earlier. If they don't, run that code again.) Try them separately.

```
~b1 = Buffer.loadDialog(s); // rerun this if you need to
~b2 = Buffer.loadDialog(s); // rerun this if you need to
(
{
   Out.ar(0,
```

```
        Pan2.ar(PlayBuf.ar(1, ~b1, loop: 1) *
              SinOsc.ar(LFNoise1.kr(12, mul: 500, add: 600)),
              0.5)
    )
}.play
)
(
{var source, delay;
    source = PlayBuf.ar(1, ~b2, loop: 1);
    delay = AllpassC.ar(source, 2, [0.65, 1.15], 10);
    Out.ar(0,
    Pan2.ar(source) + delay
    )
}.play
)
```

Figure 1.10
Buffer modulation.

1.11 Arrays, Iteration, and Logical Expressions

Instructions for playing music are often expressed as collections: a 12-tone row, a series of harmonics, a scale or mode (collection of pitches), a motive, pitch-class names, and so on. Arrays allow us to manage those collections. Earlier, we saw messages that modified an entire array, and we've used an array for stereo signals. But now we will use each item in the collection individually. To use one item in an array, we have to retrieve it. This is called *referencing* and is done using the `at` message. The argument for `at` is the index number, starting at 0. Here are some examples. Run them one at a time:

```
a = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A",   "Bb",
 "B"]; a.at(8);
"Item at index 5 is: ".post; a.at(5).postln; // why didn't   it pri
nt E?
"Item at index 0 is: ".post; a.at(0).postln; // because we   start
 with 0
do(50, {[0, 2, 4, 5, 7, 9, 11].at(7.rand).postln})
do(50, {["C", "D", "E", "F", "G", "A", "B"].at(7.rand).postln})
```

We've introduced a new message here: `do`. It is a type of iteration (see also `loop`, `while`, `for`, `forBy`) which repeats an action. The first argument for `do` is the number of repetitions (50); the second is the repeated function. A repeated process, such as a `do`,

can be placed inside a `Task` with a `wait` time. (Chapter 3 will explore timing and scheduling in more detail.)

```
Task({
   50.do({
        ["C", "D", "E", "F", "G", "A", "B"].at(7.rand).postln;
1.wait;
});
}).play // Cmd-. or equivalent to stop
```

Arrays can be used to map one type of data to another, for example, MIDI numbers and note names. Computers are better at handling numbers, but we like to see letters (C, F♯, A♭6) when working with music. A referenced array can resolve them both. Consider the random MIDI walk in figure 1.11. It picks a number between 36 and 72, but for our convenience also prints a note name using the `wrapAt` message, a variation of '`at`' that wraps numbers larger than 12 around to the beginning; 12 will return index 0, and 14 will return index 2. It also uses some simple math to print the octave.

```
(// This will not make sound, only print to the screen
Task({
   a = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A",   "B
b", "B"];
   "count, midi, pitch, octave".postln; // Print a table header   f
irst
   do(50, {arg count;
        p = rrand(36, 72);
        [count, p, a.wrapAt(p), (p/12).round(1) -1].postln;
        1.wait;
   })
}).play
)
```

Figure 1.11
Random MIDI walk.

The argument inside the `do` function (count) is similar to the arguments in the synth definition but with a critical difference: we get to name it anything we want, but we can't define what it is. It is always a counter that keeps track of the number of repetitions and is *passed* to the function by the `do`. Keeping track of iterations is very useful, as we will see later in this chapter.

We can use the `do` and `Task` to make music, stepping through a series of notes with a repeating task to play the synth (be sure you've defined PMCrotale above before

running the next example) with a wait time between each iteration. In figure 1.12, an infinite `'do'` is used (`inf.do`), but we must be careful with this. An infinite `'do'` with no wait will hang SuperCollider. Save before trying it, if you wish to do so.

```
// This patch uses the PMCrotale synth definition from figure 1.7.
Be sure it has been added (run).
(
a = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A",   "Bb",
 "B"];
"event, midi, pitch, octave".postln;
r = Task({
   inf.do({arg count;
        var midi, oct, density;
        // Density at 0.7 creates events 70% of the time.
        density = 0.7;
        // [0, 2, 4, 7, 9] is a Pentatonic scale.
        // Try other scales
        midi = [0, 2, 4, 7, 9].choose;
        oct = [48, 60, 72].choose;
        if(density.coin,
        {// true action—a note
              " ".postln;
              [midi + oct, a.wrapAt(midi), (oct/12).round(1)].   po
st;
              Synth("PMCrotale",
                  ["midi", midi + oct, "tone", rrand(1, 7),
                  "art", rrand(0.3, 2.0), "amp", rrand(0.3, 0.6),
                  "pan", 1.0.rand2]);
        },
              {["rest"]} // false action—a rest
        );
        0.2.wait;
   });
}).start
)
```

Figure 1.12
Random Crotale Walk.

Again, we've added a new convention; the logical statement `if`. The syntax for `if` is `if(condition, {true action}, {false action})`. The condition is a `Boolean` test. If the test returns a `true`, the true (first) function is executed; otherwise, the `false` function is run. Table 1.1 shows some Boolean operators. Note the important distinction

between a single equal sign, which assigns a value to a variable (myVar = 10), and two equal signs (myVar == 10), which means "Is myVar equal to 10?" Try running the examples in the true or false column alone, and you will actually see "true" or "false" in the post window. This patch uses `coin`, which takes an argument between 0 and 1.0, and will return true that percentage of the time. In this case, `0.7.coin` will be true 70 percent of the time, and `0.3.coin` will be true 30 percent of the time. It is used to control the density of events. Here, `1.0.coin` will always generate an event. Since the wait time is 0.2, which feels like a sixteenth note, the "tempo" is quarter = 75.

Table 1.1
Logical expressions

Symbol	Meaning	True Example	False Example
`==`	Equal to?	`10 == 10`	`10 == 14`
`!=`	Not equal to?	`10 != 15`	`10 != 10`
`>`	Greater than?	`10 > 5`	`10 > 15`
`<`	Less than?	`10 < 14`	`10 < 5`
`>=`	Greater than or equal to?	`10 >= 10, 10 >= 9`	`10 >= 14`
`<=`	Less than or equal to?	`10 <= 10, 10 <= 14`	`10 <= 9`
`odd`	Is it odd?	`11.odd`	`10.odd`
`even`	Is it even?	`10.even`	`11.even`
`isInteger`	Is it an integer?	`10.isInteger`	`10.2345.isInteger`
`isFloat`	Is it a float?	`10.129.isFloat`	`10.isFloat`
`and`	Both conditions	`1.odd.and(2.even)`	`1.odd.and(3.even)`
`or`	Either condition	`1.odd.or(3.even)`	`2.odd.or(3.even)`
`coin`	Return true or false	`0.7.coin` true 70% of the time	`0.7.coin` false 30% of the time

With a pulse of 0.2, several "rests" in a row will produce silences at lengths of any 0.2 multiple. An alternative method would be to remove the `if` altogether, placing the synth on its own, then choose different wait times by replacing `0.2.wait` with `wchoose([0, 0.2, 0.4, 0.8], [0.2, 0.5, 0.1, 0.2]).wait`, which chooses a value from the first array according to probabilities in the second array. In this example, that would be 0 20 percent of the time, 0.2 50 percent of the time, 0.4 10 percent of the time, and 0.8 20 percent of the time. Note that a 0 wait time means "play the next note now," ergo it is a harmonic rather than a melodic interval. Here is an isolated `wchoose` example. The `histo` message counts the number of occurrences in an array:

```
Array.fill(100, {wchoose([1, 2, 3, 4], [0.5, 0.3, 0.125,   0.07
5])}).sort.histo(4);
```

Additional `if` statements could be used for more variety; for example, to choose durations from a different range depending on an octave or MIDI note choice, setting the `dur arg` for each synth like so: `"dur", if(oct == 48, {rrand(1, 3)}, {rrand(3, 12)})`.

You might also try replacing the `1.0.rand` (pan position) with a short math calculation that will place low notes toward the left (−1), middle notes in the middle (0), and high notes on the right (1).

Here are more examples of logical expressions. The modulo `%` wraps a number so it is always within a given range. This is useful when working with scales, arrays, and counters. For example, if a variable named `ct` (for "count") runs through 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and so on, then `ct%5` will return 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, and so on:

```
if(10 == 10, {"10 is indeed equal to 10"}, {"false"})
if((1 < 20).and(1.isInteger), {"1 is less than 20"}, {"false"})
10.do({arg ea; [ea, if(ea.odd, {"odd"}, {"even"})].postln})
(
84.do({arg count; if([0, 4, 7].includes(count%12),
{count.post; "is part of a C triad.".postln},
{count.post; "is not part of a C triad".postln})})
)
50.do({if(10.rand.post > 4, {">4".postln}, {"<4".postln})})
50.do({if(10.rand > 4, {"play".postln}, {"rest".postln})})
50.do({if(0.5.coin, {"play a note".postln}, {"rest".postln})})   //
same as above
if((10.odd).or(10 < 20), {"true".postln}, {"false".postln})
```

1.12 How to do an Array

A `do` function, when attached to a number, passes a counter to the function being repeated. When `do` is attached to an array, it passes two arguments; the first is each item in the array, and the second is a counter. As before, we name them whatever we want, but they will always be each item and a counter, in that order.

```
[0, 2, 4, 5, 7, 9, 11].do({arg each, count; ["count", count,   "eac
h", each].postln})
// same
[0, 2, 4, 5, 7, 9, 11].do({arg bob, foo; [foo, bob].postln})
(
var pc;
pc = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A",   "Bb",
"B"];
```

```
[0, 2, 4, 5, 7, 9, 11].do({arg each; pc.wrapAt(each).postln;})
)
```

To illustrate *doing* an array, figure 1.13 generates a 12-tone matrix. That is (without dwelling on the details of serialism), it generates an interleaved matrix showing all 48 possible variations of a single nonrepeating series built from all 12 notes of a chromatic scale. If you know 12-tone theory (e.g., what RI6 is), you can skip the next 2 paragraphs.

```
(
var row, inversion, pitchClass;
row = Array.series(11, 1).scramble.insert(0, 0);
// or enter your own row, e.g., Webern's Op 27
// row = [0, 11, 8, 2, 1, 7, 9, 10, 4, 3, 5, 6];
row.postln;
inversion = 12-row;
// Add spaces to the pitch class strings for a tidy row
pitchClass = ["C   ", "C# ", "D   ", "Eb ", "E   ",
   "F   ", "F# ", "G   ", "Ab ", "A   ", "Bb ", "B   "];
inversion.do({arg eachInv;
   var trans;
   trans = (row + eachInv);
   // prints pitch class
   trans.do({arg step; pitchClass.wrapAt(step).post});
   " ".postln;
});
)

// More elegant with syntax shortcuts
r = ((1, 2..11).scramble.insert(0, 0).neg); r.neg.do({|i|   postln
((r + i)%12)});
(// or
p = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "T", "E"];
c = ["C ", "C#", "D ", "Eb", "E ", "F ", "F#", "G ", "Ab",   "A ",
 "Bb", "B "];
r = Array.series(11, 1).scramble.insert(0, 0);
(r.neg).do({|i| postln(p@((r + i)%12))});
(r.neg).do({|i| postln(c@((r + i)%12))});
)
```

Figure 1.13
Nested `do` to generate a 12-tone matrix.

Think of the prime version of a 12-tone set as an array. When building a matrix, the prime, or original, transpositions run from left to right. Each of those transpositions is built on successive pitches from the inversion, which occupies the first column, top to bottom. When writing out a matrix, you are in practice doing the inversion of the row to create the first column and then, at each pitch from the inversion, doing that transposition. (The difference is conceptual. Turn the matrix on its side, and the inversion becomes the original.) In this example, there are two `do` messages—one to do each pitch in I0, then a `do` for the transposition of the prime beginning on that pitch.

You might want to generate the original row randomly by scrambling an array from 0 to 11. The problem is that the prime form of a row starts with 0, and scrambling 0 through 11 would rarely place 0 at the first position. Here is a cheat; scramble 1 through 11, then insert 0 at the first index.

Figure 1.14 shows another example of additive synthesis, which blends pure sine waves (usually harmonic, but can also be inharmonic) into a single sound. This is done with an array filled with sine waves. We'll build this additive sound one sine wave at a time, each with an `LFNoise1` amplitude control. This produces random envelopes for amplitude:

```
{LFNoise1.ar(5000)}.plot // random wave
{max(0, LFNoise1.ar(5000))}.plot // return only positive values
(
{
var ampCont;
ampCont = max(0, LFNoise1.kr(12)); // slow it down for LFO   volume
control
SinOsc.ar(440, mul: ampCont)
}.scope
)
```

If we mix a bunch of sine waves, tuned to multiples (harmonics), it sounds like a very precise filter, but it is additive and would be extremely difficult to accomplish with filters. The `Mix` takes as its first argument an array of SinOsc UGens. Remove or comment out some of the SinOsc lines to better understand how it's working.

```
(
{
   var fund = 220;
   Mix.ar(
        [
              SinOsc.ar(220, mul: max(0, LFNoise1.kr(12))),
              SinOsc.ar(440, mul: max(0, LFNoise1.kr(12)))*1/2,
```

```
                SinOsc.ar(660, mul: max(0, LFNoise1.kr(12)))*1/3,
                SinOsc.ar(880, mul: max(0, LFNoise1.kr(12)))*1/4,
                SinOsc.ar(1110, mul: max(0, LFNoise1.kr(12)))*1/5,
                SinOsc.ar(1320, mul: max(0, LFNoise1.kr(12)))*1/6
        ]
    )*0.3!2
}.play
)
```

Figure 1.14
Additive synthesis.

Previously, we used `Array` to create a collection. This time, it is paired with the `fill` message (see also the `series` message) to generate an Array of UGens. The first argument is the number of items in the Array, and the second is the function used to fill the Array. It also has a counter, which we use to generate multiples of 110. (Note the patch above could have been constructed as in figure 1.15.)

```
(
{
    var nharm = 12;
    Mix.fill(nharm,
         {arg count;
                var harm;
                harm = count + 1 * 110; // remember precedence;   cou
nt + 1, then * 110
                Pan2.ar(
                    SinOsc.ar(harm,
                    mul: max(0, SinOsc.kr(count+1/4)))*1/(count+1),
                    (count/((nharm-1)/2)-1).postln)
    }
)*0.4}.play
)
```

Figure 1.15
Example of additive synthesis.

The harmonics are calculated using `count + 1*110`, but we also use the counter to synchronize the SinOsc amplitude control. These undulating LFO sine waves are controlling the amplitude of each harmonic in the series. The frequency for each increases as count increases; it is 1/4 for the first sine, 2/4 for the next, 3/4 for the next, and so on. This means that every 4 seconds we hear all 10 harmonics at full volume,

and hence a nice additive saw wave. In a way, the LFO is mimicking the additive process of each sine wave it controls. Figure 1.16 shows a sonogram of the sound.

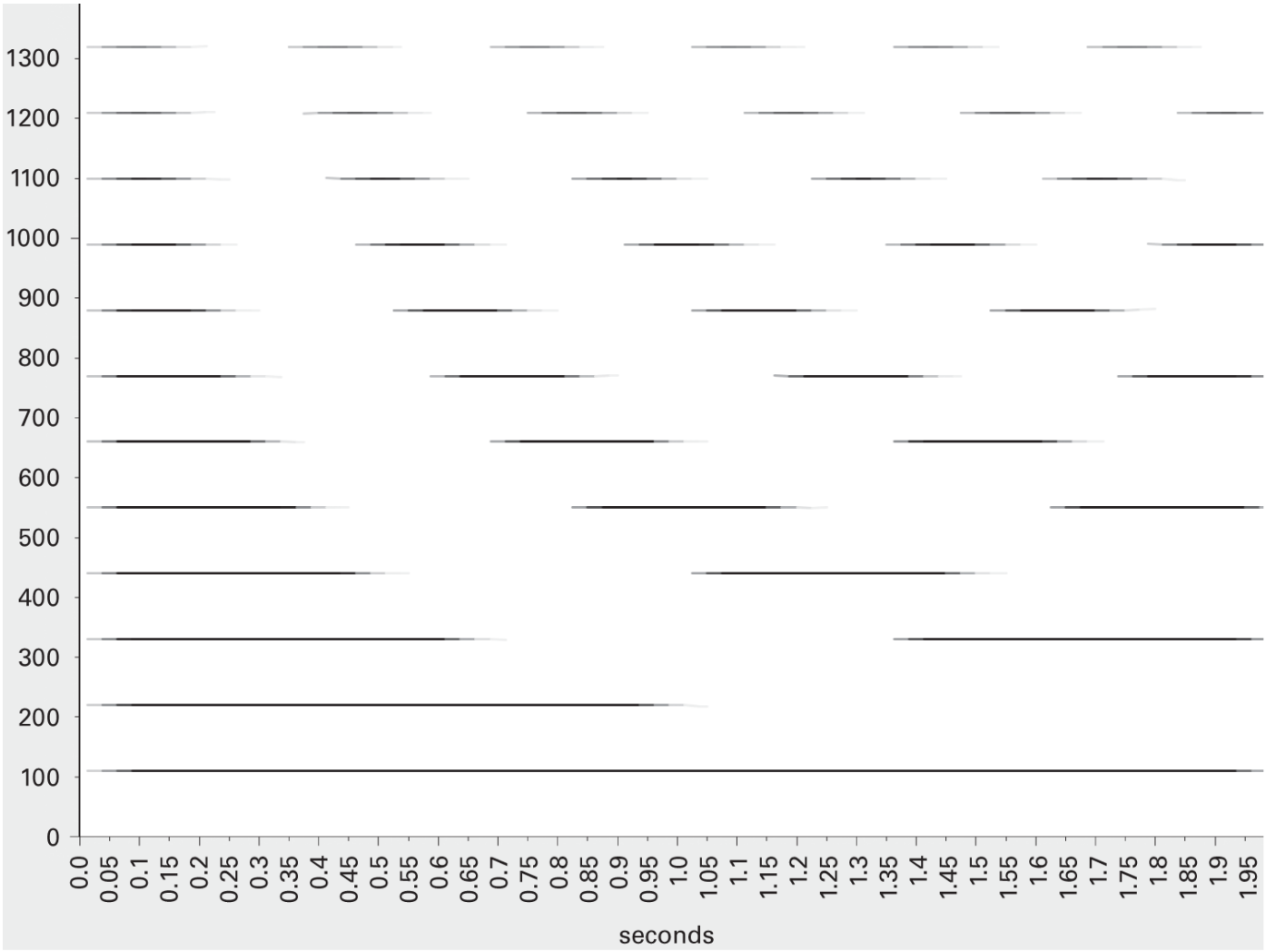


Figure 1.16
Sonogram of additive synthesis example.

Try these alternatives for the `SinOsc.ar(count+1/4)`:

```
SinOsc.kr(rrand(1/4, 5/4))
SinOsc.kr(1/4, 2pi.rand)
LFNoise1.kr(1)
LFNoise0.kr(rrand(1, 5))
```

Figure 1.17 presents a collection of bells created with `Klank`, a resonator useful for physical modeling. It requires arrays to describe the resonances, amplitudes, and rings (decay) of a virtual vibrating body. Its first argument is an array of arrays (like a box filled with boxes, each of those boxes filled with numbers). The inner three arrays are

freqs, amps, and ring times (decay). Note the tick mark (`) in front of the array: `'specs`. (Note that this is not a single quote, but a tick, or a back tick—an *accent gràve*.) We saw earlier that using an array as an argument expands the patch into different channels. We don't want that to happen here; the array needs to be mixed down to 2 channels. The tick mark keeps it from expanding into 5 channels.

```
// Try this first, to illutrate the array of arrays
Array.fill(3, {Array.fill(5, {rand(100)})})
// Then this patch
(
{
   var scale, specs, freqs, amps, rings,
   numRes = 5, bells = 10, pan;
   scale = [60, 62, 64, 67, 69].midicps;
   Mix.fill(bells, {
        arg count;
        freqs = {rrand(1, 15)*(scale.choose)}!numRes;   // frequenc
ies
        amps = {rrand(0.3, 0.9)}!numRes; // amplitudes of those   f
reqs
        rings = {rrand(1.0, 4.0)}!numRes; // decay times
        specs = [freqs, amps, rings].round(0.01); // array of   arr
ays
        // specs.postln; // uncomment to see the specs array
        pan = (count/((bells-1)/2)-1); // pan positin of each bell
        // pan = (LFNoise1.kr(rrand(3, 6))*2).softclip; // random p
an
        Pan2.ar(
              Klank.ar('specs,
                  Dust.ar(1/6, 0.03)), // excitation of model
              pan)
   })
}.play
)
```

Figure 1.17
Physically modeled bells.

1.13 Arrays in Sequences

Figure 1.18 uses arrays to play a series of sequences in minimalist style. A new variation is introduced every six iterations, or at `if(cnt1%6 == 0)`. All three variations

are reset when `cnt` gets to 24. That's how many repetitions you have to listen to, entranced, to get the full effect. There is a new convention: `amp = amp-((midiPitch-60) * 0.02);` that is, a variable assigned to the outcome a new expression, which includes its previous value. This line of code tempers the amplitude of pitches above 60, reducing them in increments of 0.02.

```
(// first define the synth with arguments
SynthDef.new("SimpleBlip", {
   arg midi = 60, tone = 10, art = 0.125, amp = 0.2, pan = -1;
   var out;
   out = Pan2.ar(
        Blip.ar(midi.midicps, tone) * EnvGen.kr(Env.perc(0.01,   ar
t)),
        pan // pan position
   );
   DetectSilence.ar(out, doneAction:2); // turn off silent notes
   // adjust volume based on midi pitch
   amp = amp-((midi-60) * 0.02);
   Out.ar(0, out*amp)
}).add;
)

(
// Simple version; pitch, tone, articulation, amp sequence
Task({
   inf.do({arg counter; // counter to advance note
        Synth("SimpleBlip", [
              \midi, [60, 61, 62, 63].wrapAt(counter),
              \tone, [1, 2, 3, 4, 5].wrapAt(counter),
              \art, [0.1, 0.2, 0.3, 0.4].wrapAt(counter),
              \amp, [0.3, 0.5, 0.6, 0.6].wrapAt(counter)*0.5,
              \pan, 0
   ]);
   0.125.wait; // tempo
   });
}).start
)

(
// More complex, using if statements to swap, replace, add, and res
et
~allThree = [0, 0, 0]; // Three containers for Blip instruments
~pSeq = [0, 0, 0]; // Three containers for sequences
```

```
~scaleAdd = [4, 5, 11, 10, 3, 6]; // Scale steps to add
~scale = [0, 2, 7, 9]; // Starting scale steps
~pitchClasses = // for converting midi numbers into pitch class
[" C"," C#"," D"," Eb"," E"," F",
   " F#"," G"," Ab"," A"," Bb"," B"];
Task({
   inf.do({
        arg thisIteration;
        var steps, durSeq, harmSeq;
        "Iteration: ".post; thisIteration.asInteger.postln;
        steps = rrand(6, 12); // Choose length of sequence
        // every fourth iteration, add a scale degree to ~scale
        // and remove that scale degree from ~scaleAdd
        if(thisIteration%4 == 3, {
             ~scale = ~scale.add(~scaleAdd.at(0));
             ~scaleAdd.remove(~scaleAdd.at(0));
        });
        // every 24th iteration, reset scales if(thisIteration%24 =
= 0,
             {~scale = [0, 2, 7, 9]; ~scaleAdd = [4, 5, 11, 10,
 3, 6];});
        "Current scale: ".post; ~scale.postln;
        // Load each pitch sequence array with choices from scale
        3.do({arg counter2; ~pSeq.wrapPut(counter2,
             Array.fill(rrand(4, 12), {
                 ~scale.choose + [48, 60].choose}))
        });
        // Print sequences
        "Each sequence in pitch class, MIDI, scale degree".postln;
        ~pSeq.do (// for each sequence
             {arg thisSequence, seqNumber;
                 "Seq ".post; (seqNumber+1).post; ": ".post;
                 // print the pitch classes and octave   thisSeque
nce.do({
                      arg thisNote; ~pitchClasses.at(thisNote%1
2).   post; thisNote.div(12).post;
                 });
                 // Print midi values and pitch class numbers
                 ", ".post; thisSequence.post; ", ".post;   (thisS
equence%12).postln;
        });
        "\n****".postln; // print return
        // fill harmonic and duration arrays with new values
```

```
        harmSeq = Array.fill(steps, {rrand(1.0, 7.0)});
        durSeq = Array.fill(steps -1, {rrand(0.01, 0.9)});
        // Stop the previous iteration of this instrument
        ~allThree.wrapAt(thisIteration).stop;
        // And start new ones
        ~allThree.wrapPut(thisIteration,
              Task({
                  inf.do({arg cnt3; // cnt2 is each iteration
                        Synth("SimpleBlip", [
                              \midi, ~pSeq.wrapAt(thisIteration).
wrapAt(cnt3),
                              \tone, harmSeq.wrapAt(cnt3),
                              \art, durSeq.wrapAt(cnt3),
                              \amp, rrand(0.1, 0.3),
                              \pan, thisIteration.wrap(-1, 2)
                        ]);
                        0.125.wait; // tempo of each note
              })}).start;
        );
        4.wait}) // time between each sequence
}).start;
)
```

Figure 1.18
Generative sequences using arrays.

The first (environment) variable is `~allThree`. It is an array used to hold each task, which plays a sequence using the defined Blip instrument. When a new sequence is executed, `wrapPut` places it in the array at the position pointed to by `thisIteration`. Before each new execution, the previous sequence in that slot is freed. An array is the best way to manage these three automated events. If we used garden-variety variables (e.g., `seq1, seq2, seq3`), then there is no way to free them. If you use `seq1.free` on one iteration, you cannot use `seq2.free` on the next. But an array can identify each iteration by way of a wrapped number, `thisIteration` (which will be 0, 1, 2): `~allThree.wrapAt(thisIteration).free`. At the fourth iteration, for example, `thisIteration` will be 3, and it will free `~allThree.wrapAt(3)`, which when wrapped is 0, or the first sequence.

Arrays are also used to manage additions to the scale, from which the sequence steps are chosen. After four repetitions, the next item in `~scaleAdd` is added to the ~scale array. Finally, although we could have used a local variable for the sequence itself (similar to `durSeq` and `harmSeq`), that would preclude resetting all three at once at

every 24th repetition (`thisIteration%24`). We have to store all three in one spot so as to access them at that point in the execution.

Each sequence is built by filling an array with between 6 and 12 (whatever value is given to steps) pitches chosen from the scale. That scale step is added to 48 (octave 4) or 60 (octave 5). Calculating the pitches separately from the octaves preserves the scale (whereas `rrand(36, 72)` would not). That array is used in the task-within-a-task, generating each pitch of the sequence using a Synth call. Using the counter `eachNote`, the task also rotates through arrays for the number of harmonics and duration (i.e., articulation). Note that the articulation array is one item shorter than the pitch array. This adds phasing to the pattern.

There are a number of variations you can try with this patch. For example, you can start with (and keep throughout) 1 scale. Or change the `scaleAdd` collection to gradually build different scales (whole tone, chromatic). Try different lengths of sequences for each element, or the same for each. Try starting and stopping each instrument at different times rather than all three at once. Try different instruments. And here is where a little planning can pay off. Note that the PMCrotale SynthDef we created earlier has the same set of arguments as the SimpleBlip SynthDef. This allows us to swap them without making any changes to the `Synth` call. Just replace `SimpleBlip` with `PMCrotale`. Try setting up another `.wait` that is less than 6 and turn off a sequence before the next one starts. Add a random pan. Add a sequence for amplitude but include values of 0 (rests).

Logical expressions are what distinguish programs like SC from user-friendly software synths; few of the latter allow for compositional algorithms such as choosing a different 12-tone row after *x* number of iterations, execution according to probabilities, or Markov chains (which require many, many arrays).

1.14 Scale and Offset

Scale is another term for a multiplier (*), and *offset* is another term for add (+). You've no doubt been pondering unexplained multipliers and adds in previous patches, the * 500 and + 1000 in figure 1.6; `carrier = LFNoise0.kr(rate) * 500 + 700`. They scale (multiply) and offset (add to or subtract from) the output of any set of values or output of UGen. Although scale and offset are just simple math, they are also, typically, the hardest ideas for beginners to grasp in this context. But they are crucial to understanding the way a UGen works.

In figure 1.19, we introduce the `poll` message and ++. `Poll` is useful for tracking values while a patch is running. We are also using a scope. Since the `SinOsc` is running at control rate, you shouldn't hear any sound. Here, ++ concatenates two strings (or arrays) into one.

```
(
{
   var trigger, wave, label, scale, offset;
   wave = SinOsc.kr(1/10); // change to 400
   scale = 1; offset = 0;
   wave = wave * scale + offset;
   label = "scale =" ++ scale.asString ++ ", offset =" ++ offse
t.asString;
   wave.round(0.01).poll(label: label);
}.scope(1)
)
```

Figure 1.19
Offset and scale.

Ten times per second, `poll` is printing the value of the `SinOsc`, rounded to the nearest multiple of 0.01. As you can see from the post window and the scope, the `SinOsc` is moving smoothly above and below 0; 1 above, −1 below, once every 10 seconds (a frequency of 1/10; i.e., 1 time in 10 seconds). A wave that oscillates between −1 and +1 is bipolar, because it deviates above and below 0.

What if we changed the `SinOsc` frequency from 1/10 to 400? Does that change the range of numbers? No; this is a common misunderstanding among beginners. The sine would still be bopping back and forth between −1 and +1, but at a much faster rate. How would that become sound? If it were an audio rate (`ar`), you could send the stream of numbers to your speakers, which would move forward (+1) and back (−1) 400 times per second, and that's what you would hear. The *range* hasn't changed, just the frequency at which it moves within that range.

It might be helpful to think of the audio wave as an array of numbers being streamed to the playback device because, in fact, that's exactly what it is. If you multiplied (*scaled*) each number in that stream by 0.5, the highest number in the range—1—would become 0.5, 0.5 would be 0.25, and so on. The lowest number would change from −1 to −0.5. All the numbers are reduced by half, and hence the speakers move forward and back half as far. In that case, the scale is being used to control volume or amplitude. Change scale = 1 to scale = 0.5, and note the difference in the wave and the numbers.

If you scale the `SinOsc` by numbers greater than 1 (5, 10, 100, 1,000), what would the output be? Of greater concern, what would that do to your speakers? Change the scale to these values and note the change. You will see the wave in the scope shoot off and above the visible portion of the window; then, as it moves to negative values, it will fly past again. Notice the stream of numbers move smoothly up to 1,000, then down to −1,000, and then repeat that range at the frequency of 1 in 10 seconds.

Change the scale back to 1 and apply an offset (or *add*) of 0.3. Note the scope still moves up and down by the same amount, but the middle of that motion is no longer 0, but 0.3. (Use the vertical zoom on the right to better see the range.) Try a scale of 100 and an offset of 600. The scope is no longer useful, but notice the numbers. The deviation is 100, and the center value of the deviation becomes 600. That is, the stream of numbers is starting at 600 (instead of 0), then moving up 100 to 700, then down 100 below 600, to 500. In short, the wave was 0, plus and minus 1; now it's 600, plus and minus 100. It's just math.

Replace the `SinOsc` with `LFSaw` and `LFNoise1`, and carefully watch the results. Each will have the same range of values, but one is in the shape of a ramp and the other is random.

Values this high are not appropriate for amplitude unless you're using your speakers for booster rockets. But they are useful as controls. You didn't hear any effect in this patch because the output is sent to a control bus, which we don't hear.

Why would we scale and offset an oscillator that high? Where might we use a smooth, low-frequency sine motion, ramp motion, or random motion in the range between 500 and 700 in a patch? As we hope is obvious, you would use this as a control for another UGen's frequency input, where values that high are expected. This is illustrated in figure 1.20. It uses two `SinOsc` UGens. One is generating the frequency we hear (frequencies between 500 and 700); the other, an LFO (low-frequency oscillator), is providing numbers for the outer `SinOsc` frequency argument. This controls the undulation above and below 600 Hz like a slow, wide vibrato. In addition to the poll, which monitors the value of `abs(control)`, we've attached two scope messages to the control UGen and the audio UGen.

```
(
{
   var trigger, control, scale, offset;
   // try other values, but not greater than offset scale = 300;
   offset = 600; // try other values
   trigger = Impulse.kr(10);
   control = SinOsc.ar(1/4).scope("control"); // LFO
   control = control * scale + offset;
   SinOsc.ar(freq: abs(control).poll).scope("audio")
}.play
)
```

Figure 1.20
SinOsc offset and scaled for control.

Change the frequency of the control oscillator from 1 in 4 (1/4) to 1, 2, or 5, for instance—maybe higher for FM? Change the control UGen (`SinOsc`) to other waveforms, such as `LFNoise1`, `LFNoise0`, `LFSaw`, `LFTri`, and `LFPulse`. Change the scale and offset, noting how they change the sound. See if you can adjust the scale and offset to create a convincing vibrato.

A warning about offset and scale: some UGens will produce unpredictable sounds with negative values (e.g., `CombN`). A simple error in calculating the scale and offset can yield unwanted negative values. There are two simple methods for avoiding this: never use a scale larger than the offset or nest the control in an `abs` function.

Offset and scale are used to set a UGen's output to the correct range for controls sent to other UGens. To find the correct range, you have to make this calculation: What range of input do we need for this control? What is the controlling UGen's default output? Therefore, how do we have to scale and offset the default to get the range we want? Try it with the index of a `PMOsc`. An effective range is 1 to 15. To control it with an `LFNoise1`, which has a default output of 0, +/−1, you would offset by 8 to make that the center of the range, then scale by 7 so that the range would be 8, +/− 7 (1 to 15). This is simple enough, but not all UGens are bipolar. Many (envelopes, lines, LFPulse) are unipolar, with a default output of 0 to 1. (The Help file usually tells you which they are, or you can use the signalRange method to find out, e.g., `SinOsc.ar.signalRange`.) To control the same index with a range of 1 to 15, the `LFPulse` would be scaled by 14 and offset by 1 ((0 to 1) * 14 is 0 to 14, then + 1 is 1 to 15).

The following explanations may help to clarify.

For bipolar UGens: The offset is the center of the range, and the scale is the distance it deviates from the center. For example, an offset of 1000 and a scale of 200 means the range will be 200 above and below 1000 (1000 +/− 200), which is 800 to 1200. Also, the scale is half the range. The offset is the lowest value of the range, plus the scale. If you want a range of 10 to 20, the scale is 5 (half the range), and the offset is 15 (low value plus the scale, or center of the range).

For unipolar UGens, the offset is the lowest value in the range, and the scale is the range. The scale plus the offset is the highest value. For example, an offset of 500 and a scale of 1,000 means that the range will be 500 to 1,500 (500 + 1,000).

Table 1.2 shows some examples matching control type, example range, the most common UGen defaults (bipolar −1–0 and unipolar 0–1), scale, and offset, and how to keep them straight.

Table 1.2
Scale and offset examples

Parameter	Example Range	Default	Offset	Scale	Think of it as

Parameter	Example Range	Default	Offset	Scale	Think of it as
MIDI	48–72	Bipolar, −1–1 Unipolar 0–1	60 48	12 24	60 +/− 12 48 + 24
Amplitude	0–0.8	Bipolar, −1–1 Unipolar 0–1	0.4 0	0.4 0.8	0.4 +/− 0.4 0 + 0.8
Frequency	200–1,800	Bipolar, −1–1 Unipolar 0–1	1000 200	800 1,600	1,000 +/− 800 200 + 1,600
Harmonics	1 to 23	Bipolar, −1–1 Unipolar 0–1	12 1	11 22	12 +/− 11 1 + 22

That was a lot to take in, but it's essential to everything that we will do—so essential that most UGens have arguments for both scale and offset, called `mul` and `add`, respectively. Since these arguments are so common, they usually aren't documented in Help files.

There are some shortcuts. The `range` message will map both unipolar and bipolar UGens to the range specified by their low and high arguments. So `SinOsc.ar.range(500, 1000)` and `LFPulse.ar.range(500, 1000)` will both generate values between 500 and 1,000, even though `SinOsc` is bipolar and `LFPulse` is unipolar. If you use range, however, make sure that you don't supply values for `mul` and `add`, as that would result in a conflict. There are other similar methods documented (along with `scope`, `poll`, etc.) in the UGen Help file, which is a good file to get familiar with.

Figure 1.21 shows a more complicated example using a classic sample and hold, which strikes a pleasing balance between repetition and variety. Some control ranges are set using `.range`, some with `mul` and `add`, so that each is appropriate for the parameter that it is being used to control.

```
(
a = SynthDef("Latch_demo",
    {
        arg rate = 9;
        var freq, latchrate, index, ratio, env, out;
        latchrate = rate*LFNoise0.kr(1/10).range(1.3, 1.9);
        index = Latch.kr(
              LFSaw.kr(latchrate).range(4, 5),
              Impulse.kr(rate)
        );
        freq = Latch.kr(
              LFSaw.kr(latchrate,
                  mul: max(0, LFNoise1.kr(1/5).range(-14, 34)),
                  add: LFNoise0.kr(1/7).range(48, 72)),
              Impulse.kr(rate)
```

```
        ).round(1).midicps;
        ratio = LFNoise1.kr(1/10).range(3.0, 7.0);

        env = EnvGen.kr(
              Env.perc(0, LFNoise0.kr(rate).range(0.5, 2.5)/rate),
              Impulse.kr(rate),
              LFNoise1.kr([5, 5], 2, 1).max(0).min(0.8));
        out = PMOsc.ar(
              [freq, freq * 1.5],
              freq*ratio,
              index,
              mul: env
        );
        Out.ar(0, out);
    }
).play
)
```

Figure 1.21
PMOsc with sample and hold (latch).

1.15 When Bad Code Happens to Good People (Debugging)

Even after years of coding, you will still encounter the occasional error message. When it happens, and it will, try these recovery strategies: proofread, undo, multiple versions, monitor, selectively evaluate, comment out suspect code, and use "safe" values. Let's start with a working patch (figure 1.22) and then introduce some errors. It has a random trigger, so you may have to listen a dozen or so seconds before the first attack strikes. Run it a few times to start several bells.

```
(
{// it's just a bell
    var burst, burstEnv, bell, delay, dry,
    burstFreq = 500, freqs, amps, rings;
    burstEnv = EnvGen.kr(Env.perc(0, 0.05),
         Dust.kr(1/5), 0.1);
    // burstEnv.poll(100, "env");
    burst = SinOsc.ar(freq: burstFreq,
         mul: burstEnv);
    // burst.poll(100, "burst");
    freqs = Array.fill(10, {exprand(100, 10000)});
    amps = Array.fill(10, {rrand(0.01, 0.1)});
```

```
    rings = Array.fill(10, {rrand(1.0, 6.0)});
    // [freqs, amps, rings].round(0.01).postln;
    // "safe" values
    // freqs = [100, 200, 300, 400];
    // amps = [1, 1, 1, 1];
    // rings = [1, 1, 1, 1];
    bell = Pan2.ar(
         Klank.ar('[freqs, amps, rings], burst),
         rrand(-1.0, 1.0)
    );
    delay = AllpassN.ar(bell, 2.5,
         [LFNoise1.kr(7, 1.5, 1.6), LFNoise1.kr(7, 1.5, 1.6)],
         1, mul: 0.8);
    bell
    + delay
    // + SinOsc.ar(mul: LFPulse.kr(1) * 0.05);
}.play
)
```

Figure 1.22
It's just a bell.

Introduce the following errors into figure 1.22. First, remove the `.kr` from the first `EnvGen.kr` and run it. Notice that this error doesn't even offer a message; the patch runs, but you don't get any sound. This illustrates the most common error: a typo. The first thing that you should do, before tearing a patch apart, is to proofread. Another common typo is an unmatched enclosure (e.g., {without}, [without], or "without a closing"). Try removing any of the enclosures and executing the code. You'll see plenty of these innocuous but annoying parse errors. But typos can cause further errors.

As you develop a patch, you may introduce half a dozen changes (as we did here, intentionally) into something that ran fine prior to the last evaluation. Then, suddenly, it's broken. To find out which change caused the crash, you can back out of the error, as we have been doing with this patch, and "undo until it's unbroken." Retrace your steps by repeating Undo/Execute until it is working again. When the error disappears, choose Redo. The code that caused the error will be selected. Select Edit/Undo as many times as necessary to restore the original patch, then change `Env.perc` to `Env.parc` and evaluate. These longer, more cryptic messages can be difficult to decipher at first, but they can offer clues even to beginners, especially if you look near the top, where there is a higher density of comprehensible words; you will see "parc not understood," pointing to the typo.

Another strategy is to save multiple versions. When a patch is working the way that you like, save it, then immediately choose Save As and rename it (e.g., "myPatch 1.13"). You can experiment with this new file worry free. If something breaks, revert to the working version.

One can also proof by following a variable through the patch, noting the values as you read. For this, you may have to use twentieth-century writing devices (a pencil and paper). Write two or three variables and record what happens to each variable as you proofread through your code. A common error is to use a variable before it is given a value. As an example, change `burstFreq = 500` to just `burstFreq` (remove the `= 500`). When that variable arrives at `SinOsc`, it has a value of `nil`, causing an error. (Nil is a special value in SC that means "nothing" or "empty".)

You can also monitor values using `postln`. If you put the post on a new line with a collection of values (in an array) such as `[freqs, amps, rings].postln`, it can be turned on and off by commenting it out, but you can also sneak in a `postln` at lots of places. For example,

```
Array.series(12, 1, 1).scramble.insert(0, 0)
```

can become

```
Array.series(12, 1, 1).postln.scramble.postln.insert(0, 0).postln
```

The post can even be made conditional during a repeated process, such as `do`: `if(count%9 == 0, {[count, myVar].postln})`. (If you are repeating a process, monitor the counter too. This will tell you at what step the program failed.)

Similarly, you can isolate and evaluate small sections of code. In the example above, select `Array.series(12, 1, 1)` only and press [enter]. Next add `.scramble`, then `.insert`. If the results aren't what you expected or wanted, then the message is not what the program expected or needed.

The `poll` message can also be used to monitor a UGen's values in real time. The first two arguments are rate (how often the UGen is polled) and label (a string that accompanies the posted value). We've included two commented examples in figure 1.22. In addition, the scope can be attached to any UGen, as seen in many examples. (See also the `Poll` UGen.)

You can isolate which part of a patch is causing an error by using comments to disable suspect lines, but this can be tricky. In figure 1.22, you may suspect the three lines beginning with `delay =`. But if you comment out those lines, `bell + delay` will introduce a new error because you've removed delay. You would have to comment out

both the `delay` chunk and `+ delay`, or give `delay` a safe value, such as 0. This, however, may also cause its own set of errors.

Try replacing `exprand(100, 1000)` with 0. This effectively kills the patch. Loading arrays with a complex expression may give unwanted results. To isolate this kind of problem, one can use safe values (i.e., values which are known to work). For example, if we suspect the lines that fill freqs, amps, and rings, we can insert safer lines (commented) below. When we uncomment the lower three lines, they replace the upper three lines, pointing to the faulty section. Indeed, we may develop the patch this way, first putting it together with simple, safe values, then gradually adding more complex processes while keeping the original to fall back on when things go wrong.

A test tone is a type of safe value. For example, you can uncomment the `+ SinOsc` at the bottom. This adds a pulse-controlled test tone that beeps on and off to make sure the patch is indeed running and making sound (in case the rest of the code is indeed working, just not creating usable audio), thus helping to narrow down what went wrong. If you hear the test tone but not the bells, then the issue is probably not in the code itself (like a typo) but a conceptual error (e.g., a signal at 0 amplitude).

A more wily and destructive gremlin is code that returns `NaN` (not a number) or `inf` (infinite) values. What you'll hear is a single pop, followed by no sound from that patch or any other patch, even a simple, isolated `{SinOsc.ar}.play`. In this case you may need to quit the server: `s.quit` will do this in code. Restart with `s.boot`. (Or use the corresponding items on the Server menu.) You can also quit and relaunch SC.

Remove all the errors we introduced into our example and try adding some monitoring points using `postln` and `poll`. Of course, you need not wait for a disaster for an excuse to poke around. Use these techniques to figure out what is going on in a working patch. Much can be revealed in the examples from this and other chapters with a few well-placed `post, scope,` and `Poll` UGens.

Finally, not all errors are errors. Sometimes aberrant values produce interesting results. Exploration in SC can yield serendipitous rewards for the adventurous programmer.

1.16 What Next?

Take a moment to sit back and take in what you've learned. SC is a collection of virtual synthesis modules that can be linked through nesting or with variables to create complex sounds. Using the Help files, you can learn what objects there are and what messages and arguments they understand. You can match the output of one module to the input of another with scale and offset, creating automated controls. These patches can be sent to the server as SynthDefs, with arguments that can be changed for real-time composition and performance. Or you can record the patches (using the Start Recording item under

the Server menu) to be used in a DAWstyle composition. Finally, you've learned some strategies to overcome crashes.

Now that you've done some actual code examples, you will see the value of shortcuts. You'll find a complete list of shortcuts in the Help documentation.

You should also start writing your own personal Help file. When you generate a clever patch or solve a complex problem, paste it into your personal Help code file for future reference.

Read this chapter again. Many of the examples were, out of necessity, more complicated than the accompanying explanation. Now more things will make sense. Although there will still be nebulous sections (by design, encouraging you to dig on your own), you no doubt will be pleasantly surprised at how much more you understand the second time through.

Peruse the Help files. Try the examples. Reverse-engineer them (and these examples). Put them together one piece at a time. This is a time-honored way of learning. And, of course, read the other chapters in this text, written by a cadre of international SC developers.

Join the online community. Answer someone else's question. It's good karma.

Finally, have fun. In the early days of synthesis development, one had to go where the equipment was. This often meant getting accepted in a degree program at a distant university, and then finding an apartment and moving all your junk to that city just to compete with a dozen other composers for a few hours a week over one machine. You, on the other hand, don't have to be a D.M.A. candidate, and you don't have to move to another town. You can pull out your laptop right now. The worst you'll get are startled pets, a hard reboot from an infinite `do`, or annoyed glances from a resident significant other or nearby library patrons (even when using headsets). You'd be crazy not to fiddle with it every spare minute. You are obligated to take advantage of such a golden opportunity. Tell your significant other/mom/boss/cat/iguana that we gave you permission, and get to it.

Notes

1. 24

```
[5, 5.123] (both numbers and brackets)
Entire LFSaw line
1
0.4
1 and 0.3
Everything between the 2 curly brackets.
```

2. `{PMOsc.ar(Line.kr(10, 1000, 60), Line.kr(1000, 10, 60), Line.kr(1, 30, 60))}.play`

3.

```
{
var rate = 4, carrier, modRatio, ind;
rate = Line.kr(start: 1, end: 12, dur: 60);
carrier = LFNoise0.kr(freq: rate) * 500 + 700;
ind = Line.kr(start: 1, end: 12.0, dur: 60);
modRatio = MouseX.kr(minval: [1, 1], maxval: 2.0);
PMOsc.ar(carfreq: carrier, modfreq: carrier*modRatio,
  pmindex: ind)*0.3
}.play
```

2 The Unit Generator

Joshua Parmenter

2.1 Introduction

As you learned in chapter 1, Unit Generators (UGens) are used for generating and processing signals with the SuperCollider synthesis server (hereafter referred to by its application name, scsynth). This chapter will cover the basic functionality of UGens in the standard SuperCollider distribution, as well as some of the more common extension libraries. As SuperCollider is a descendant of the Music N family of synthesis programs, UGens in SuperCollider provide an efficient modular system for creating complicated audio networks, which we'll refer to as UGen graphs (Roads, 1996). Examples will demonstrate the efficient creation of SynthDef functions, as well as how to group UGen graphs together for dynamic synthesis and audio processing controls. In addition, the examples will offer hints of optimizations that maximize processor efficiency for real-time performance. Event triggering, system debugging, and signal routing will also be demonstrated.

In some cases, I will discuss more technical aspects, such as details of the source code for UGens, or use specific technical terms from the field of digital audio. If you've arrived at this chapter as a beginner (having just completed chapter 1), don't worry if not everything immediately makes complete sense. Just trust that some of the information provided is for more technically savvy users, and come back to it later if you need to. In some cases, I've provided more basic explanations in the notes.

2.2 UGens, scsynth, and SuperCollider

The UGen is the signal-generating and -processing workhorse for the SuperCollider synthesis environment. I use the term "environment" specifically to immediately bring to light the role that UGens themselves play, where they actually exist in the program, and how they are represented inside the SuperCollider language. UGens are defined within libraries, which are written in the C++ programming language and compiled before use. These libraries (plug-ins) are loaded into an instance of scsynth when it is booted and persist until the server quits. Any sound produced by scsynth is the result of a UGen, and UGens are used only inside scsynth. UGens are also represented in the

SuperCollider language's class system. If you've not encountered classes in programming before, a class is a sort of template that makes objects like UGens or Arrays when they are sent the appropriate message. For example, in the code `SinOsc.ar`, the class is `SinOsc` and `ar` is the message. (This topic is explored in more detail in chapter 5.) It is important to realize that the UGen class itself does not return signal values to the language. The signal output of a UGen is generally not available to the language unless values are specifically sent from the running synthesis server to the language (usually through network messaging). As a result, the role of UGens in the SuperCollider language is restricted to the composition of SynthDefs for the description of a UGen graph. These SynthDefs will be sent to the server, and only *there* can they produce sound.

A number of methods exist to create SynthDefs for you; for example, `Function:play` (represented in the language as `{}.play`) will wrap a signal flow description into a SynthDef. Although some of these language constructs may make it look as though synthesis is happening inside the language, it is important to keep in mind that all synthesis is done on the server, and a SynthDef is always created. To avoid confusion, I will simply refer to both the SuperCollider language objects and the server-side signal-processing units as "UGens."[1]

The chief job of the majority of SuperCollider's UGens is the generation of an output signal, but what happens inside the UGen to generate the signal will vary widely. SuperCollider computes this output in blocks of values (called "samples") for both audio rate (`ar`) and control rate (`kr`) signals (i.e., calculating several values at a time for each UGen in turn). The default block size is 64 samples, with an ar UGen computing 64 samples and a kr UGen returning a single value for each block. The terms "block" and "control period" should therefore be understood as closely related. The input parameters to a UGen will typically be a scalar value (i.e., a number), a control rate signal, or an audio rate signal. Some UGens require a specific kind of input, such as `BufRd`, where the phase argument must be the same rate as the BufRd UGen itself. Special cases like this are usually noted in the Help file for a given UGen. In cases where a UGen's input is limited to the control rate, audio rate signals are still allowed, but only the first sample in the block will be read. Internally, control rate signals are usually interpolated linearly in cases where a noninterpolated signal would cause noise in the resulting signal. Linear interpolation means that the signal will ramp up from the previous control rate value to the value that was sampled at the beginning of the control period.[2] In doing this, it will change its value by a constant amount for each sample in the block and would thus appear as a straight line if you graphed the signal. (This is the *linear* aspect.) In cases where an audio rate input may be more common, the UGen's source code function will check the input parameter's rate and run the appropriate function for you.[3]

Before moving on to a more in-depth discussion of the input and output of a typical UGen, I'll briefly expand upon and review some of the discussion from chapter 1, starting with the very common `mul` and `add` parameters. UGens whose output will be used as an audio signal typically output samples with a range of floating-point values between 1.0 and −1.0 (i.e., a *bipolar* range, as noted earlier). For reasons of efficiency, `mul` and `add` are implemented by wrapping a UGen inside a separate `MulAdd` UGen. `MulAdd` takes the output of a UGen, scales its values by `mul`, and adds `add`. Then `mul` defaults to 1.0, and add defaults to 0.0. Although the typical UGen's output is +/−1.0, in reality, this works out as +/−mul + add. Using this, output values can have a broad range, so a UGen's output can be much higher than 1.0 or lower than −1.0.[4] Since the `mul` and `add` parameters are so common, they will usually be left out of the discussion of a particular UGen's input.

The input to a UGen parameter may also be an Array of values or UGens. As noted in chapter 1, in these cases, a UGen will "multichannel expand." Figure 2.1 shows how an Array affects and expands the output of a SinOsc UGen, as well as ways to manipulate the output Array that results. Using the SynthDef \UGen_ex1a, a single SinOsc will be generated and its output will be sent to audio bus 0. If all you have done is start the server and run the scope, your server window should show 12 UGens running. Once \UGen_ex1a is run, you should see 15: the original 12 plus 1 UGen for the SinOsc, 1 UGen for the MulAdd that the SinOsc is wrapped in, and 1 UGen for the Out UGen. The SynthDef \UGen_ex1b shows the multichannel expansion that results from the use of an Array with 4 elements for the `freq` parameter. The output is spread over output channels 0 through 3, and there are now 9 UGens that are used in this SynthDef. Out accounts for 1 UGen, 4 more for the 4 SinOscs, and 4 more for the MulAdd UGens. The use of Arrays is one of the simplest ways in SuperCollider to create multichannel sound, though each element of the Array is simply output to the corresponding channel. (In other words, this isn't really sophisticated panning.) (See chapter 14 for a detailed discussion of multichannel synthesis and spatialization.)

```
s = Server.local.boot;
z = s.scope(4);
// a) mono output
(
a = SynthDef(\UGen_ex1a, {
   Out.ar(0, SinOsc.ar(440, 0, 0.1))
}).play(s);
)
a.free;
// b) freq input is an Array of 4 items—outputs to busses 0-3
(
```

```
a = SynthDef(\UGen_ex1b, {
   Out.ar(0, SinOsc.ar([440, 446, 448.5, 882], 0, 0.1))
}).play(s);
)
a.free;
// c) Array is added to the 'mul' arg to show mapping
(
a = SynthDef(\UGen_ex1c, {
   Out.ar(0, SinOsc.ar([440, 446, 448.5, 882], 0, [0.1, 0.2,   0.3,
0.4]))
}).play(s);
)
a.free;
// d) The output of the SinOsc above is actually an Array of
// four SinOscs. Sum them together for an additive   synthesis exam
ple.
(
a = SynthDef(\UGen_ex1d, {
   Out.ar(0, SinOsc.ar([440, 446, 448.5, 882], 0, [0.1, 0.2,   0.
3]).sum);
}).play(s);
)
a.free;
z.window.close;
```

Figure 2.1
Arrays and multichannel expansion.

In the SynthDef \UGen_ex1c, a 4-element Array is used for the `mul` argument. As a result, the SinOsc created with the frequency located at index 0 in the `freq` Array will be joined with the element at index 0 in the `mul` Array, the elements at index 1 are paired, and so on. No new UGens are added to this example (since the 4 `MulAdds` were already present in the previous SynthDef). Part d shows 2 concepts, the first exploring what happens when Arrays of different sizes are used. Elements 0, 1, and 2 will match up between the 2 Arrays, with element 3 in the `freq` Array paired with element 0 from the `mul` Array. Second, the sum method, when applied to an Array, will return a single value representing the sum of the Array's values, in this case the summing of the output of the 4 SinOscs contained inside the Array. Due to the way that SynthDefs are optimized, there is no additional UGen as a result of the sum method! In the course of building the SynthDef, basic addition like this will be folded into the MulAdd UGens, taking advantage of the addition that already exists.

As mentioned in chapter 1, the output signals of UGens can be used as inputs to another UGen. The output of other synthesis nodes (Synths) themselves can also be accessed. The In UGen will allow access to data written to an audio or control bus. It should be noted that the order of execution for nodes and the difference between audio and control signals need to be taken into account for these signals to be used correctly. To fully understand the power of the In and Out UGens, the nature of the bus system needs to be addressed first.

The number of audio buses available for use is set up at the server's boot time, using its ServerOptions. By default, 1,024 audio buses are allocated. Of these, the output buses to the system hardware are numbered first, as set by the numOutputBusChannels parameter in ServerOptions. By default, this number is 2, corresponding to audio buses 0–1. The next block of buses is set to read audio from the system inputs and correspond in number to the numInputBusChannels value. This value again defaults to 2 and corresponds to channels 2–3. The remaining buses can be used for any routing that is not intended for direct input or output to the system, acting like virtual buses in standard audio sequencer software. Even though system input buses are set aside for reading in sound from the computer, these buses, like any other bus, can be written to. The same can be said for the output buses; there is no reason that the signal written to them cannot be read and further manipulated. It is important to understand that, in a technical sense, buses are nothing more than small areas of memory where samples can be stored. In this sense, they are rather like Arrays in the language, which can also be used to store a collection of values. Audio buses are large enough to hold one block's worth of samples; control buses contain a single value.

Every time a block of samples is produced, scsynth will calculate sample values according to its node tree. The node tree gives the running order for working through the Synths; if Synth B's input depends on Synth A's output, it is important to calculate A before B. Every Synth on the server exists as a node in this tree. The other type of node is the Group, which is used to organize a set of Synths into a particular subset. (In computer science terms, Groups are internal nodes of a tree and the Synths are the leaves.)

For each synthesis node in the tree, scsynth will evaluate the corresponding SynthDef's code line by line. In practice, this means calculating the output of each UGen in turn, in the order they are specified. If you are using one or more `Out` UGens in your synthesis process (or if one is created for you, using `Function.play`, for instance), any specified output from the UGen graph is written to an audio bus. The next node in the tree is then evaluated in the same way. If audio from a previous node has already been written into an audio bus, any node that follows later in the node tree can access it with an `In` UGen. Through arguments to both the Synth and Group objects, as well as convenience methods such as `after` and `before`, you are able to place nodes or groups

of nodes anywhere you want in the node tree's order of execution. You can have the server print out its current node tree—showing each group and synth in order, labeled by id and SynthDef name—by executing `s.queryAllNodes`, through a Server menu item or click on the Server status bar in the standard IDE, or by selecting a Server window created with `s.makeWindow` and pressing "n." (See the Synth, Group, and Order of Execution Help files for more detail on this topic.)

Before any synthesis nodes are calculated for a given block cycle, all audio buses have their samples zeroed out (i.e., they are reset to silence). Incoming signal is then filled in to the hardware input buses. When an `Out` UGen is used, signal written to a bus is added to the current signal already on the bus. `Out` also has some variants. A `ReplaceOut` UGen will zero out any signal on the bus before writing its output, and the `XOut` UGen will write a mixture of the UGen's output with what is already on the audio bus according to an `xfade` (cross-fade) argument. `OffsetOut` is a special-case Out UGen that can start writing output in the middle of a control block and should be used when sample accurate timing is necessary.

You can write to and read from the control buses with the kr versions of In and Out, respectively.[5] Unlike the audio buses, control bus values are not zeroed out and persist across control blocks. They thus change only when touched. If more than one value is written to a control bus within the same control period, the values are added together; otherwise, older values are replaced. There are no control buses associated with system hardware, and since they contain only a single sample, they have a much smaller overhead. By default, 16,384 control buses are set up when the Server boots. Examples of using the control buses to control multiple Synths will be shown near the end of the chapter, in the section on optimization tips and tips for taking advantage of the server's order of execution in the node tree.

In addition to signals that are internal to a SynthDef and those that are routed through the In UGen, many UGens are also able to access, write, and share data through the use of memory buffers. As shown in chapter 1, buffers may contain sound file samples but also may be used to store any other data, such as recorded signals, FFT frames, or analysis data from a number of other programs.

Now that the basics of the input and output for UGens have been covered, a more in-depth exploration of the tools available can take place, as well as a discussion of creative ways to optimize your usage of UGens.

2.3 UGens Available in SuperCollider

The first place that anyone should look when starting to use SuperCollider is the Tour_of_UGens Help file, which is accessible from the main Help page. The tour is a mini-tutorial on signal creation and processing. This Help file is a wonderful resource

for beginners to the SuperCollider language. Even after years of synthesis and DSP experience, I still find new ideas in the basic examples in this file. I make it a point to browse it when I am stuck or between pieces. Although there is very little in the examples that should be used directly, the techniques and use of the SuperCollider language demonstrated here make it one of the gems of the Help file system.

Without going into great detail, the tour of the UGens file will show you many of the basic signal generators that are built into SuperCollider. Oscillators with both band-limited and non-band-limited shapes (i.e., limited or unlimited in terms of the frequency spectra of their output) are available. Also included are a number of table lookup oscillators—UGens which derive their output by looping through a predetermined table of sample values—that access both internal tables (this is the case with SinOsc and its built-in sine table) and user-supplied tables that allow for the creation of complex periodic signals through the buffer system (the `Osc` and `OscN` UGens, for example). Buffers can also be accessed for sound file playback (`PlayBuf`, `BufRd`), and sound can be written to memory in real time (using `RecordBuf` and `BufWr`, as well as a number of buffer-based delay UGens). A wide variety of signal-processing UGens are also available, from the most basic filters (`FOS`, `SOS`, `OnePole`, and `OneZero`), through user-friendly standard filters[6] (`LPF`, `BPF`, `HPF`, `Resonz`, `MoogFF`, for example), to more complex delay- and feedback-based processes. (The Comb family of delays and `Pluck` are examples.) An `FFT`/`IFFT`-based phase vocoder system is also available. SuperCollider has a large system of random-number generators for the creation of random values, ranging in rate from a single sample to periodically interpolated values and sample-by-sample generators for noises of different colors (e.g., white, pink, etc.). The tour also shows examples of numerous distortion techniques that map trigonometry functions onto a signal, various clipping techniques, and advanced waveshaping available through the `Shaper` UGen.

A number of UGens are also available for spatialization. These typically return an Array of signals corresponding to the number of outputs for a given panner, from the simple equal-power stereo panning of `Pan2`, through the `PanAz` UGen for panning across, and up to numerous audio channels. `PanB2` and `DecodeB2` provide the means for 2-dimensional Ambisonic panning, and a number of extension libraries expand these capabilities to fill 3-dimensional Ambisonics through higher orders. (Again, see chapter 14 for more information.)

UGens are also available for smoothing control signals (e.g., `Lag` and `Decay` smooth using different shapes). A number of UGens also exist for the creation of envelopes (`Line`, `XLine`, and the `EnvGen` UGens, in combination with `Env` to specify an envelope itself). Although the tour covers most of the techniques just mentioned and provides easy access to the Help files for each UGen shown, the remainder of this chapter will deal with notable omissions from the tour Help file.

One of the questions most commonly asked by new users deals with dynamic processing. Figure 2.2 shows the use of a `Compander` (SuperCollider's general-purpose dynamics processor) for compressing an incoming signal. Compander is a hard-knee processor that compares the input `src` signal against the input `control` signal, and then adjusts the src according to the slopes that are described by the remaining parameters. This means that the gain can be scaled up or down, depending on whether the input is higher or lower than the control, resulting in *compression* or *expansion* of the dynamic range of the signal. There are parameters for the amount of time used for gain adjustment to take effect on signals that are out of range, as well as for the amount of time that the UGen takes for the gain adjustment to relax. Since the attack is so hard, the signal, even when passed through the Compander, may overshoot the maximum output allowed for a 32-bit signal (+/−1.0 in linear amplitude), which can result in clipping (a kind of distortion). To prevent clipping, the output of the Compander is then fed into a fast look ahead `Limiter`. Using the `poll` method (which is very handy for debugging), the peak outputs of the compressed values, the compressed and limited values, and the source (`src`) peak are tracked and printed to the post window to show the difference in the output values. Since Limiter introduces a delay, the compressor and src signal are also delayed, so all the signals line up with the Limiter output.

```
Server.default = s = Server.local.boot;
z = s.scope;
(
SynthDef(\UGen_ex2, {arg freq = 440;
   var src, compressor, limiter, out;
   // 10 SinOsc's, mixed together. Output amplitude is controlled
 // with an Dust UGen wrapped in a Decay2 UGen to create a // spike
with an Exponential Decay
   src = SinOsc.ar(
        // a harmonic series based on freq
        Array.series(10, freq, freq),
        0, // phase
        Array.fill(10, {Decay2.ar(
              // Dust will create an impulse about every 2 seconds,
// with values between 0 and 5
              Dust.ar(0.1, 5),
              // Decay2, attach time of 0.01 seconds and a decay //
time of 5 seconds to allow for a build up of // signal
              0.01, 5)});
        ).sum;
   // compress signal about 0.5
   compressor = Compander.ar(src, src, 0.5, 1, 0.1);
```

```
    limiter = Limiter.ar(compressor, 0.5);
    // out is the compressed only signal on the left, the // compres
sed and limited on the right
    out = [DelayN.ar(compressor, 0.02, 0.02), limiter];
    // use Peak and poll to track the highest output values. // Upda
tes every second
    Peak.ar(out ++ src, Impulse.kr(1)).poll(1, ["compressed",   "lim
ited", "src"]);
    Out.ar(0, out);
}).add;
)
a = Synth(\UGen_ex2, [\freq, 440]);
a.free; z.window.close;
```

Figure 2.2
Dynamics processing.

Figure 2.3 shows one of the uses for triggers inside SuperCollider. In general, a trigger is any signal on the server that changes from a value of 0 or less to a positive value. In the example, the trigger is used to control the `GrainFM` UGen. A new grain is created every time a trigger occurs from the signal fed into the trig input parameter. UGens such as `Impulse` (outputs single samples of value `mul` at a periodic rate, and 0 the rest of the time) and `Dust` (random impulses with a density parameter) are excellent for controlling trigger-based UGens. For GrainFM, as well as the other granular synthesis UGens that come with SuperCollider (`TGrains`, `GrainSin`, `GrainBuf`, and `GrainIn`),[7] the rate of the trigger is important. In this case, audio rate trigger UGens will create sample accurate grains, whereas a kr input will create new grains only at the beginning of a block. When a trigger is received by the UGen, the remaining inputs are polled to control each grain's parameters. For more on granular synthesis and SuperCollider, see chapter 16.

```
(
SynthDef(\UGen_ex3, {arg gate = 1, amp = 1, rate = 10;
    var trigger, dur, carfreq, modfreq, index, pan, env;
    trigger = Impulse.ar(rate);
    dur = rate.reciprocal;
    carfreq = LFNoise2.kr.range(100, 110);
    modfreq = LFTri.kr(0.1).exprange(200, 840);
    index = LFCub.kr(0.2).range(4, 10);
    pan = WhiteNoise.ar.range(-0.1, 0.1);
    env = EnvGen.kr(
```

```
            Env([0, 1, 0], [1, 1], \sin, 1),
            gate,
            levelScale: amp,
            doneAction: 2);
        Out.ar(0,
            GrainFM.ar(2, trigger, dur, carfreq, modfreq, index,
                  pan, −1) * env)
}).add;
)
a = Synth(\UGen_ex3, [\rate, 80, \amp, 0.2]);
b = Synth(\UGen_ex3, [\rate, 42, \amp, 0.2]);
c = Synth(\UGen_ex3, [\rate, 121, \amp, 0.2]);
[a, b, c].do({arg thisSynth; thisSynth.set(\gate, 0)});
```

Figure 2.3
Triggering from within the server.

Before going on to the use of triggers for sending values from the server to the language client, the `range` and `exprange` messages in the above example deserve mention. We've already seen in chapter 1 that the range method can provide a shortcut to setting the range of a UGen's output by computing the necessary `mul` and `add` values for both bipolar (normal output between −1.0 and +1.0) and unipolar (output between 0.0 and 1.0) UGens. Range does this in a *linear* fashion (i.e., in a "straight line," as described above in the discussion about interpolation). The use of `exprange` on the `LFTri` UGen in figure 2.3 will map the linear output values between 0.0 and 1.0 to an exponential curve between 200 and 440. (Exponential mapping is very useful when dealing with a UGen that is controlling frequency, since it is closer to the way our ears perceive it.) In this case, not only has the range of the signal changed, but so has the shape!

As noted at the beginning of this chapter, UGens run on the server, and it is not possible for the SuperCollider language to access their output directly. The `SendTrig` UGen represents one of the few ways for scsynth to send information back to the client language.[8] An `OSCdef` must be set up inside the language to look for the `/tr` message sent by the SendTrig UGen. The OSC in OSCdef is short for Open Sound Control, which is the network protocol SC uses to communicate between the language and the server. An OSCdef is thus something that responds to an incoming OSC message. The OSCdef function will be evaluated when a trigger message is received. (For a more detailed discussion of OSC and OSCdef, see chapter 4.) It should be noted that these data are communicated through the network and are not suitable for sending audio data from the server to the language. However, a large amount of useful information can still be communicated using a number of signal analysis and binary operation UGens

available. Many of the Boolean operations available in the language (which test if something is true or false, such as greater than or less than) will return 0 when false, and 1 when true, when applied to UGens and can then be used as a trigger signal. The UGen `InRange` will tell you if a signal's current value is between 2 values. UGens that take a trigger, such as `Timer` and `Latch`, can also have their output polled at the time of the trigger, and these values can then be sent back to the language. Figure 2.4 shows a number of SynthDefs that will analyze the audio from the first audio input to the system. We'll use the `SoundIn` UGen to get the input signal (which is just a shorthand that saves us having to offset to the first hardware input bus), and then we'll send information about it back to the language with various triggers. The OSCdef function checks for various id values that are sent back with the trigger information and will provide a unique auditory response to the right speaker based on this information. I recommend using headphones and giving the input some sort of musical input. Be varied with pitch, rhythm, and force of attack. In order to limit the number of triggers that are sent by each process, the `Trig` UGen is used. In case you need to send back more complicated information to the language, you can use the `SendReply` UGen, which allows for sending back Arrays of values at the same time and lets you use a custom label (e.g., myArrayMessage).

```
(
SynthDef(\UGen_ex4a, {arg id, limit = 1;
   var src, pitch, hasPitch, keynum, outOfTune;
   // read input
   src = SoundIn.ar(0);
   // analyze the frequency of the input
   #pitch, hasPitch = Pitch.kr(src);
   // convert to a midi keynum, but don't round! This value // will
be used later.
   pitch = pitch.cpsmidi;
   // if you are within an eighth tone of an equal tempered // pitc
h, send a trigger
   outOfTune = (pitch-pitch.round).abs < 0.25;
   // if outOfTune is true, send a trigger. Limit to 1 trigger // e
very 'limit' seconds
   SendTrig.kr(Trig.kr(outOfTune, limit), id, pitch.round);
}).add;

SynthDef(\UGen_ex4b, {arg id1, id2, limit = 1, thresh = 0.5;
   var src, amp, amptrig, timer;
   src = SoundIn.ar(0);
   // analyze the amplitude input, cause a trigger if the // output
```

```
is over the thresh
   amp = Amplitude.kr(src);
   amptrig = Trig.kr(amp > thresh, limit);
   // use amptrig to see how long it is between triggers.
   timer = Timer.kr(amptrig);
   // send the values back with two different ids
   SendTrig.kr(amptrig, id1, amp);
   SendTrig.kr(amptrig, id2, timer);
}).add;

// plays a SinOsc of the pitch you were closest to
SynthDef(\UGen_ex4c, {arg freq;
   Out.ar(1, SinOsc.ar(freq, 0, XLine.kr(0.1, 0.00001,   0.5, doneA
ction: 2)))
}).add;

// modulated noise to respond to amp spikes
SynthDef(\UGen_ex4d, {arg freq;
   Out.ar(1, LFNoise1.ar(200) * SinOsc.ar(freq, 0,
   XLine.kr(0.1, 0.00001, 0.5, doneAction: 2)));
}).add;

// allocate three unique ids for the trigger ids
a = UniqueID.next;
b = UniqueID.next;
c = UniqueID.next;

// an envelope to poll for amp values later
e = Env([440, 880], [1], \exp);

// add the responder
o = OSCdef(\test, {arg msg, time, addr, recvPort;
   // the msg is an array with 4 values. . .post them
   msg.postln;
   // the id sent back from the SendTrig is msg[2]. . .use // it to
decide what to do
   case
        // pitch trigger
        {msg[2] == a}
        // msg[3] is the rounded keynum
        {Synth(\UGen_ex4c, [\freq, msg[3].midicps])}
        // amp trigger
        {msg[2] == b}
```

```
        // play a noise burst, higher the amp value, higher the //
 freq (polls the Env 'e')
        {Synth(\UGen_ex4d, [\freq, e[msg[3]]])}
        // use the Timer value to play a delayed noise burst at //
 2000 Hz
        {msg[2] == c}
        {SystemClock.sched(msg[3], {
              Synth(\UGen_ex4d, [\freq, 2000]);
        })}
}, '/tr').enable;

// schedule the start our listening synths. . .
// then sing or tap away on the input.
SystemClock.sched(1.0, {
   Synth(\UGen_ex4a, [\id, a, \limit, 1]);
   Synth(\UGen_ex4b, [\id1, b, \id2, c, \limit, 0.2, \thresh,   0.2
5]);
});
// add a command period function to stop the synths and remove // t
he responder
CmdPeriod.doOnce({
   OSCdef(\test).clear; "Removed the responder".postln;
})
)
```

Figure 2.4
Triggers and sending values from the server back to the SuperCollider language.

Generating random values on the server, like any other process, needs to be handled with the use of UGens. The `SimpleNumber` methods such as `rand` and `rrand` do not work properly inside a SynthDef since they will be evaluated by the language only once, when the SynthDef is built, rather than in a continuous manner once they are running on the server. The `WhiteNoise` UGen will give you a new random value for every sample, with a range of +/−mul (similar to the `rand2` method in the language) and is suitable for generating a full-spectrum noise signal. At the other extreme are the `Rand` UGen, which creates a single random floating-point value, and `IRand`, which will create a single integer value upon the instantiation of a Synth. `TRand` and `TIRand` are *k*-rate UGens that will generate a new random value whenever a trigger occurs. There are also UGens for generating values with different distributions (see `LinRand`, `ExpRand`, and `NRand`). The `LFNoise` and `LFDNoise` UGens are excellent for slower-moving, random signals, providing a number of interpolation methods between each random value. Figure 2.5 shows sample uses of a number of these UGens, as well as how randomness in a Synth can be seeded.[9]

```
(
SynthDef(\UGen_ex5, {arg gate = 1, seed = 0, id = 1, amp = 1;
   var src, pitchbase, freq, rq, filt, trigger, env;
   RandID.ir(id);
   RandSeed.ir(1, seed);
   env = EnvGen.kr(Env([0, 1, 0], [1, 4], [4, -4], 1), gate,   done
Action: 2);
   src = WhiteNoise.ar;
   trigger = Impulse.kr(Rand.new(2, 5));
   pitchbase = IRand.new(4, 9) * 12;
   freq = TIRand.kr(pitchbase, pitchbase + 12, trigger).midicps;
   rq = LFDNoise3.kr(Rand.new(0.3, 0.8)).range(0.01, 0.005);
   filt = Resonz.ar(src, Lag2.kr(freq), rq);
   Out.ar(0, Pan2.ar(filt, LFNoise1.kr(0.1)) * env * amp)
}).add;
)
a = Synth(\UGen_ex5, [\seed, 123]);
a.release;

// Using the same seed, we get the same gesture
b = Synth(\UGen_ex5, [\seed, 123]);

b.release;

// passing in different seeds
(
r = Routine.run({
   thisThread.randSeed_(123);
   10.do({
        a = Synth(\UGen_ex5, [\seed, 10000.rand.postln, \amp,   3.d
bamp]);
        1.wait;
        a.release;
   })
});
)
```

Figure 2.5
Random-number generators and random seeding in the server.

`FreeVerb` and `FreeVerb2` (for stereo inputs) and `GVerb` (an implementation of Juhana Sadeharju's reverb algorithm) are also included in the standard SuperCollider

distribution. The FreeVerb multichannel expands like any other UGen (and can be useful for wrapping around a Pan UGen's output to localize the reverberation). Both FreeVerb2 and GVerb output stereo. In the case of GVerb, input is mono, and a spread parameter controls the width of the reverb field. FreeVerb2 takes two inputs, corresponding to the left and right channels. These three UGens also have a built-in low-pass filter for high-frequency damping within the reverb feedback loop. GVerb adds controls for reverb time, as well as level controls over the dry, early, and tail portions of the signal. Like most other reverbs, each of these UGens has its own color, so tinkering with their controls is essential. Because of the controls given in the GVerb UGen, a number of unusual and interesting effects, beyond what you might expect from a reverb, are possible (figure 2.6).

```
(
SynthDef(\UGen_ex6, {arg gate = 1, roomsize = 200, revtime = 450;
   var src, env, gverb;
   env = EnvGen.kr(Env([0, 1, 0], [1, 4], [4, -4], 1), gate,   done
Action: 2);
   src = Resonz.ar(
            Array.fill(4, {Dust.ar(6)}),
            1760 * [1, 2.2, 3.95, 8.76] +
                Array.fill(4, {LFNoise2.kr(1, 20)}),
            0.01).sum * 30.dbamp;
   gverb = GVerb.ar(
        src,
        roomsize,
        revtime,
        // feedback loop damping
        0.99,
        // input bw of signal
        LFNoise2.kr(0.1).range(0.9, 0.7),
        // spread
        LFNoise1.kr(0.2).range(0.2, 0.6),
        // almost no direct source
        -60.dbamp,
        // some early reflection
        -18.dbamp,
        // lots of the tail
        3.dbamp,
        roomsize);
   Out.ar(0, gverb * env)
}).add;
)
```

```
a = Synth(\UGen_ex6);
a.release;
```

Figure 2.6
GVerb.

There are many more useful and powerful UGens available as extensions to the SuperCollider server. The main SuperCollider web page (<https://supercollider.github.io/>) provides links to these resources. SuperCollider forum and code sharing sites, as well as the historic sc-users mailing list archive, also provide an enormous resource for UGens as they are developed. If you can't find something you need, asking the community is a good first step. The UGen may already exist, or you may find a developer who is bored at 2 A.M. on a Saturday and willing to do it for you!

2.4 Optimization Tips

UGens are generally the most CPU-intensive part of SuperCollider, with the possible exception of fast-changing GUI processes. Whereas GUI processes are purposely run on a lower-priority thread in the operating system to avoid audio glitches, UGens on scsynth run at the highest priority. This means that although a GUI process may take up lots of CPU, if the server needs the CPU to process audio, the GUI process will take a back seat. But once the server is using all the CPU time available, there is little that the computer can do, and your audio will glitch. Though the UGens themselves tend to be highly optimized, there are still things that you as a user can do to minimize overhead.

The first tip is to use *k*-rate UGens whenever possible. If you are using a UGen to control a parameter rather than for actual audio output, using the *k*-rate version can sometimes significantly reduce your CPU usage with no noticeable effect on your sound. Figure 2.7 shows an example of this, where a slow-moving `LFNoise2` UGen is used to control the panning of a SinOsc. In the first SynthDef, the UGen is run at audio rate and shows an average CPU usage of about 56 percent on my machine. Switching the LFNoise2 UGen to kr reduces the load to about 39 percent. There are probably few optimizations that will reduce your load this highly, but in real-time situations where CPU is becoming an issue, it is worth looking through your SynthDefs to see where you may be using a-rate UGens that aren't needed. If you are working on a fixed piece with scsynth's NRT (non-real-time) mode (see chapter 18), kr UGens can still be very useful for quicker renders while testing. Then, when you are ready to do a final render, the ServerOptions blockSize parameter can be set to 1, in effect turning all of the kr UGens into audio rate ones.[10] This, of course, will slow compilation time greatly, but your signals will be free of any interpolation noise that would be introduced with the larger control blocks.

```
(
SynthDef(\UGen_ex7a, {arg gate = 1, freq = 440, amp = 0.1,   rate =
0.2;
   var src, pos, env;
   src = SinOsc.ar(freq, 0);
   pos = LFNoise2.ar(rate);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, levelScale: amp,   d
oneAction: 2);
   Out.ar(0, Pan2.ar(src, pos) * env);
}).add;

SynthDef(\UGen_ex7b, {arg gate = 1, freq = 440, amp = 0.1,   rate =
0.2;
   var src, pos, env;
   src = SinOsc.ar(freq, 0);
   pos = LFNoise2.kr(rate);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, levelScale:   amp, d
oneAction: 2);
   Out.ar(0, Pan2.ar(src, pos) * env);
}).add;

SynthDef(\UGen_ex7c, {arg gate = 1, freq = 440, amp = 0.1,   rate =
0.2;
   var src, pos, env;
   src = SinOsc.ar(freq, 0);
   pos = LFNoise2.kr(rate);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, levelScale:   amp, d
oneAction: 2);
   Out.ar(0, Pan2.ar(src * env, pos));
}).add;

)
// 17% on my machine
(
a = Group.new;
250.do({
   Synth(\UGen_ex7a, [\freq, 440.0.rrand(1760.0), \amp, 0.001,   \r
ate, 0.2], a)
});
```

```
)
a.release;

// 12%
(
a = Group.new;
250.do({
   Synth(\UGen_ex7b, [\freq, 440.0.rrand(1760.0), \amp, 0.001,   \r
ate, 0.2], a)
});
)
a.release;

// 13%
(
a = Group.new;
250.do({
   Synth(\UGen_ex7c, [\freq, 440.0.rrand(1760.0), \amp, 0.001,   \r
ate, 0.2], a)
});
)
a.release;
```

Figure 2.7
Audio rate and control rate comparisons.

The final optimization in this example has to do with the placement of the enveloping multiplier. Note that in the first two SynthDefs, the output of Pan2 is multiplied by the envelope. Since Pan2's output is an Array of two outputs, the EnvGen also multichannel expands, so two EnvGens are actually used! In the third SynthDef, the mono src is multiplied before it is panned, saving additional overhead. (Note also that the number of UGens to run the last example has fallen from 2,000 to 1,750.) And all of the Synths in this case are using the same envelope shape. Through inter-Synth signal routing, this load could be reduced even more.

Signal routing can also reduce CPU usage, as well as give you global control over a number of Synths. Figure 2.8a shows a sound similar to that in figure 2.7, this time with Ambisonic encoding and decoding.[11] Since each Synth is decoding the Ambisonic signal in the same way and using the same envelope, these 2 aspects can be incorporated into their own SynthDef. All the encoding Synths will be placed into a single Group. The Ambisonic signal will be patched from each individual Synth into a single Synth that will capture all of the output from the Group, and decode and envelop the resulting sound. The UGen usage is reduced from 2,250 to 1,257, and the CPU usage from 57

percent to 38 percent. The trick here is to remember to use the correct `addAction` for the catchall Synth; by adding it *after* the source Group, we ensure that order of execution is correct. (As you may have gathered, Groups are an excellent way of organizing and ordering your Synths.) This approach to signal routing can also reduce your CPU load when using a limiter or reverb on a large number of Synths, or indeed in any case where a single process can be applied to multiple source Synths. At the same time this approach lets you make a level or parameter change to your synthesis process in a single location rather than in multiple Synths.

```
(
SynthDef(\UGen_ex8a, {arg gate = 1, freq = 440, amp = 0.1,   rate =
0.2;
   var w, x, y, out, env, decode;
   #w, x, y = PanB2.ar(
        SinOsc.ar(freq, 0), LFNoise2.kr(rate));
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, levelScale: amp,   d
oneAction: 2);
        decode = DecodeB2.ar(2, w, x, y);
        Out.ar(0, decode * env)
}).add;

SynthDef(\UGen_ex8b, {arg outbus, freq = 440, rate = 0.2;
   var w, x, y;
   #w, x, y = PanB2.ar(
        SinOsc.ar(freq, 0), LFNoise2.kr(rate));
   Out.ar(outbus, [w, x, y])
}).add;

SynthDef(\UGen_ex8c, {arg inbus, gate = 1, amp = 0.1;
   var w, x, y, env, decode;
   #w, x, y = In.ar(inbus, 3);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, levelScale:   amp, d
oneAction: 14);
   decode = DecodeB2.ar(2, w, x, y) * env;
   ReplaceOut.ar(0, decode);
}).add;
)

(
a = Group.new;
```

```
250.do({
   Synth(\UGen_ex8a, [\freq, 440.0.rrand(1760.0), \amp, 0.001,   \r
ate, 0.2], a)
});
)
a.release;

(
a = Group.new;
z = Bus.audio(s, 3);

// the 'catch-all' synth for decoding and enveloping
Synth(\UGen_ex8c, [\inbus, z, \amp, 0.001], a, \addAfter);   // add
it after the Group containing the encoding synths
250.do({
   Synth(\UGen_ex8b, [\freq, 440.0.rrand(1760.0), \outbus, z,   \ra
te, 0.2], a)
});
)
a.release;
```

Figure 2.8
Signal routing optimizations.

Figure 2.9 shows how certain math operations may benefit from computation on the client side rather than on the server. In the SynthDef \UGen_ex9a, the values that are passed into the Synth are in dB, and the SynthDef takes care of converting the value inside the SynthDef into linear amplitude. However, in every control block this calculation is done on a value that is set only once, when the Synth starts. The second SynthDef expects linear amplitudes, with the conversion done in the language when the parameter is passed to the Synth. The CPU usage in these examples decreases from 45 percent to 36 percent. If the value is updated later from the language, you just need to make the conversion only when setting the new value. However, some UGens (such as Rand in these examples) evaluate only once, when the Synth starts. Doing the calculation in the language doesn't yield a significant difference in CPU usage.

```
(
// pass in amp in db
SynthDef(\UGen_ex9a, {arg gate = 1, freq = 440, amp = 0;
   var src, env;
   src = SinOsc.ar(freq, 0, amp.dbamp);
   env = EnvGen.kr(
```

```
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   Out.ar(0, Pan2.ar(src * env, Rand(−1.0, 1.0)));
}).add;

// pass in linear amplitude
SynthDef(\UGen_ex9b, {arg gate = 1, freq = 440, amp = 1;
   var src, env;
   src = SinOsc.ar(freq, 0, amp);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   Out.ar(0, Pan2.ar(src * env, Rand(−1.0, 1.0)));
}).add;

SynthDef(\UGen_ex9c, {arg gate = 1, freq = 440, amp = −3,   pos =
0;
   var src, env;
   src = SinOsc.ar(freq, 0, amp);
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   Out.ar(0, Pan2.ar(src * env, pos));
}).add;
)

// 57% on my machine
(
a = Group.new;
250.do({
   Synth(\UGen_ex9a, [\freq, 440.0.rrand(1760.0), \amp, −60], a)
});
)
a.release;

// 38%
(
a = Group.new;
250.do({
   Synth(\UGen_ex9b, [\freq, 440.0.rrand(1760.0), \amp,   −60.dbam
p], a)
});
)
a.release;

// 38% (no difference from b)
```

```
(
a = Group.new;
250.do({
   Synth(\UGen_ex9c, [\freq, 440.0.rrand(1760.0), \amp,   -60.dbam
p, \pos, 1.0.rand2], a)
});
)
a.release;
```

Figure 2.9
Client-side calculations versus server-side calculations.

One final thing to look out for is UGens with different kinds of interpolation. A number of delay UGens, as well as variable-rate buffer UGens, offer different methods of interpolation. The Comb UGen and its N, L, and C variants are a good example. N, L, and C refer to no interpolation, linear interpolation, and cubic interpolation, respectively.[12] One might imagine that you should always use the `CombC` UGen because the cubic interpolation version will always give you the "best" delay. If the delay amount is one that would land between samples, this is certainly true. If the delay time is changing, the change of pitch that results will benefit from a smooth, accurate interpolation. However, especially in instances where the delay length is fixed, it will be worth your time to hear if linear interpolation is close enough for your ears, or even if any interpolation is needed at all. Figure 2.10 gives a couple of examples. In it, SynthDefs \Ugen_ex10a, \Ugen_ex10b, and \Ugen_ex10c all use a dynamic delay time with the 3 different interpolation methods, and the difference in sound is noticeable. If a large number of these UGens were used, a CPU difference would also be seen. For comparison, SynthDefs \Ugen_ex10d, \Ugen_ex10e, and Ugen_ex10f use a fixed delay length with the three different interpolation methods. For feedback delays that result in a resonant pitch (in this example, the pitch "e" should resonate), a difference can be noticed. However, with larger delay times, the sonic difference between the interpolation methods is negligible, if it can be noticed at all.

```
(
SynthDef(\UGen_ex10a, {arg gate = 1;
   var src, delay, env;
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
   delay = CombN.ar(src, 0.1, Line.kr(0.0001, 0.001, 10));
   Out.ar(0, (delay * env).dup);
}).add;
```

```
SynthDef(\UGen_ex10b, {arg gate = 1;
   var src, delay, env;
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
   delay = CombL.ar(src, 0.1, Line.kr(0.0001, 0.001, 10));
   Out.ar(0, (delay * env).dup);
}).add;

SynthDef(\UGen_ex10c, {arg gate = 1;
   var src, delay, env;
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
   delay = CombC.ar(src, 0.1, Line.kr(0.0001, 0.001, 10));
   Out.ar(0, (delay * env).dup);
}).add;
)

a = Synth(\UGen_ex10a); // no interpolation
a.release;

a = Synth(\UGen_ex10b); // linear interpolation
a.release;

a = Synth(\UGen_ex10c); // cubic interpolation
a.release;

(
SynthDef(\UGen_ex10d, {arg gate = 1, deltime = 0.001;
   var src, delay, env;
   env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
   src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
   delay = CombN.ar(src, 0.1, deltime);
   Out.ar(0, (delay * env).dup);
}).add;

SynthDef(\UGen_ex10e, {arg gate = 1, deltime = 0.001;
   var src, delay, env;
   env = EnvGen.kr(
```

```
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
    src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
    delay = CombL.ar(src, 0.1, deltime);
    Out.ar(0, (delay * env).dup);
}).add;

SynthDef(\UGen_ex10f, {arg gate = 1, deltime = 0.001;
    var src, delay, env;
    env = EnvGen.kr(
        Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction: 2);
    src = Decay.ar(Impulse.ar(1), 1.0, PinkNoise.ar(0.1));
    delay = CombC.ar(src, 0.1, deltime);
    Out.ar(0, (delay * env).dup);
}).add;
)

// tune to a specific pitch
a = Synth(\UGen_ex10d, [\deltime, 100.midicps.reciprocal]);   // no
interpolation
a.release;

a = Synth(\UGen_ex10e, [\deltime, 100.midicps.reciprocal]);   // li
near interpolation
a.release;

a = Synth(\UGen_ex10f, [\deltime, 100.midicps.reciprocal]);   // cu
bic interpolation
a.release;

// a much longer delay
a = Synth(\UGen_ex10d, [\deltime, 0.1]); // no interpolation
a.release;

a = Synth(\UGen_ex10e, [\deltime, 0.1]); // linear interpolation
a.release;

a = Synth(\UGen_ex10f, [\deltime, 0.1]); // cubic interpolation
a.release;
```

Figure 2.10
Interpolation methods in delays, and the processor and sonic differences.

2.5 Conclusion

UGens are probably the most actively developed area of the SuperCollider program. The fact that SuperCollider gets its signal-processing power from dynamically loaded libraries allows developers and users to create, share, and explore new DSP and signal-creation approaches quickly and easily. This aspect of the program is also wonderful for researchers looking for a flexible way of exploring and manipulating how we perceive sound. It is the creative combination of these basic tools, however, that gives the artist using SuperCollider a large amount of freedom, as well as the ability to build the tools needed to create sounds, textures, and structures that don't yet exist.

Notes

1. It is possible to communicate with the Server from other applications (called "clients") than from the SuperCollider language. I trust that readers who do use other applications will be able to make the distinction between the language classes and UGens on the server.
2. This linear interpolation is one of the features of SuperCollider that I think make it sound so clean. Many other synthesis programs that have a control rate use a sample-and-hold approach to control rate values, keeping a single value for the entire block and possibly introducing more noise into the system than a linearly interpolated signal does.
3. As an example, the `SinOsc` UGen has 2 inputs, one for the `freq` argument and another for `phase`. Both of these inputs can take an audio rate signal (allowing for frequency and phase modulation) or a control rate signal. As a result, inside the source code for the SinOsc UGen, there are 4 different functions to handle each specific combination of inputs. One function (labeled in the source next_aa) will compute output values for the SinOsc if both `freq` and `phase` are audio rate. The next_ak function will handle the case where `freq` is `ar` but `phase` is `kr` or slower, and the next_ka handles the reverse case. Finally, next_kk is used when both inputs are `kr` or slower. However, it is not uncommon for most UGens to create different calculation functions for every input possibility, especially since many `kr` inputs would be linearly interpolated. Therefore, in cases where an ar input to a signal does not seem to be responding as you expect it to, this should be one of the first limitations to consider. It could be that the UGen reads the input only at the control rate, and the `ar` signal is simply having its first sample in each block period taken.
4. The technical name for SC's internal sample format is linear amplitude 32-bit floating point. This allows for a huge range of values.
5. Control buses can also have values written to them with the OSC commands "/c_set" and "/c_setn" and the methods Bus:set and Bus:setn. See the Server-Command-Reference and Bus Help files for more information.
6. Filters remove part of a signal's frequency spectrum. For instance, LPF stands for "low-pass filter," which means it "passes" frequencies below a specified cutoff unchanged and removes them to an increasing extent as you go above the cutoff. HPF stands for "high-pass filter," which does the opposite. The Tour of UGens file has examples of these which will make this clearer.
7. In addition to the granular UGens mentioned here, a number of third-party UGens are available as extensions by both myself and Bhob Rainey. Bhob's granular UGens are variants of the TGrain UGen that allow for different windowing functions. My granular UGens also offer different windowing functions, as well as the ability to interpolate between two different tables containing different envelopes. I also have variants that support Ambisonic output.
8. Poll also broadcasts a "/tr" message and can be used in a similar way.
9. Randomness in computers is never really random; it just seems that way. Technically, it's referred to as "pseudo-random." Because of this, it is possible to "seed" a Synth's random-number generator with a value. The practical upshot of this is that if you do, you will always get the same sequence of pseudo-random values if you provide the same seed.
10. It should be noted that some UGens react badly to block sizes other than 64 samples. For example, many machine-listening ones are dependent on keeping the ServerOption's blockSize at the default 64.
11. Ambisonics is a multichannel spatialization technique. See chapter 14 for further discussion.
12. The Comb UGens are a type of delay and involves combining a signal with a delayed version of itself, which results in a kind of filtering. The interpolation here refers to the method used to derive the delayed signal if the delay

time does not correspond to an integer multiple of samples.

Reference

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

3 Composition with SuperCollider

Scott Wilson and Julio d'Escriván

3.1 Introduction

The actual process of composing, and deciding how to go about it, can be one of the most difficult things about using SuperCollider. People often find it hard to make the jump from modifying simple examples to producing a full-scale piece. In contrast to Digital Audio Workstation (DAW) software such as Pro Tools, for example, SC doesn't present the user with a single "preferred" way of working. This can be confusing, but it's an inevitable side effect of the flexibility of SC, which allows for many different approaches to generating and assembling material. A brief and incomplete list of ways people might use SC for composition could include the following:

- Real-time interactive works with musicians
- Sound installations
- Generating material for tape music composition (to be assembled later on a DAW), perhaps performed in real time
- As a processing and synthesis tool kit for experimenting with sound
- To get away from always using the same plug-ins
- To create generative music programs
- To create a composition or performance tool kit tailored to one's own musical ideas.

All of these activities have different requirements and suggest different approaches. This chapter attempts to give the composer or sound artist some starting points for creative exploration. Naturally, we can't hope to be anywhere near exhaustive, as the topic of the chapter is huge and in some senses encompasses all aspects of SC. Thus we'll take a pragmatic approach, exploring both some abstract ideas and concrete applications, and referring you to other chapters in this book where they are relevant.

3.1.1 Coding for Flexibility

The notion of making things that are flexible and reusable is something that we'll keep in mind as we examine different ideas in this chapter. As an example, you might have some code that generates a finished sound file, possibly your entire piece. With a little

planning and foresight, you might be able to change that code so that it can easily be customized on the fly in live performance or be adapted to generate a new version to different specifications (quad instead of stereo, for instance).

With this in mind, it may be useful to utilize environment variables which allow for global storage and are easily recalled. You'll recall from chapter 1 that environment variables are preceded by a tilde (~):

```
// some code we may want to use later. . .
~something = {Pulse.ar(80)*EnvGen.ar(Env.perc, doneAction: 2)};
// when the time comes, just call it by its name and play it!
~something.play
```

Since environment variables do not have the limited scope of normal variables, we'll use them in this chapter for creating simple examples. Keep in mind, however, that in the final version of a piece there may be good reasons for structuring your code differently.

3.2 Control and Structure

When deciding how to control and structure a piece, you need to consider both practical and aesthetic issues: Who is your piece for? Who is performing it? (Maybe you, maybe an SC Luddite …) What kind of flexibility (or expressiveness!) is musically meaningful in your context? Does pragmatism (i.e., maximum reliability) override aesthetic or other concerns (i.e., you're a hard-core experimentalist or you are on tenure track and need to do something technically impressive)?

A fundamental part of designing a piece in SC is deciding how to control what happens when. How you do this depends upon your individual needs. You may have a simple list of events that need to happen at specific times or a collection of things that can be triggered flexibly (for instance, from a GUI) in response to input from a performer, or algorithmically. Or you may need to combine multiple approaches.

We use the term "structure" here when discussing this issue of how to control when and how things happen, but keep in mind that this could mean anything from the macro scale to the micro scale. In many cases in SC, the mechanisms you use might be the same.

3.2.1 Clocks, Routines, and Tasks

Here's a very simple example that shows you how to schedule something to happen at a given time. It makes use of the `SystemClock` class.

```
SystemClock.sched(2, {"foo".postln;});
```

The first argument to the `sched` message is a delay in seconds, and the second is a `Function` that will be evaluated after that delay. In this case, the Function simply posts the word "foo," but it could contain any valid SC code. If the last thing to be evaluated in the Function returns a number, SystemClock will reschedule the Function, using that value as the new delay time:

```
// "foo" repeats every second
SystemClock.sched(0, {"foo".postln; 1.0});
// "bar" repeats at a random delay
SystemClock.sched(0, {"bar".postln; 1.0.rand});
// clear all scheduled events
SystemClock.clear;
```

SystemClock has one important limitation: it cannot be used to schedule events which affect native GUI widgets on OSX. For this purpose, another clock exists, called `AppClock`. Generally, you can use it in the same way as SystemClock, but be aware that its timing is slightly less accurate. There is a shortcut for scheduling something on the AppClock immediately, which is to wrap it in a Function and call `defer` on it.

```
// this causes a "You can not use this Qt functionality in the curr
ent thread. Try scheduling on AppClock instead." error (Qt is the u
nderlying GUI library that sclang uses)
SystemClock.sched(1, {Window.new.front});
// defer reschedules GUI code on the AppClock, so this works
SystemClock.sched(1, {{Window.new.front}.defer});
```

The term "GUI" refers to things such as windows, buttons, and sliders. This topic is covered in detail in chapter 12, so although we'll see some GUI code in a few of the examples in this chapter, we won't worry too much about the finer details of it. Most of it should be pretty straightforward and intuitive, anyway, so for now, just move past any bits that aren't clear and try to focus on the topics at hand.

Another Clock subclass, `TempoClock`, provides the ability to schedule events according to beats rather than in seconds. Unlike the clocks we've looked at so far, you need to create an instance of TempoClock and send sched messages to it, rather than to the class. This is because you can have many instances of TempoClock, each with its own tempo, but there's only one each of SystemClock and AppClock. By varying a TempoClock's tempo (in beats per second), you can change the speed. Here's a simple example:

```
(
t = TempoClock.new; // make a new TempoClock
```

```
t.sched(0, {"Hello!".postln; 1});
)
t.tempo = 2; // twice as fast
t.clear;
```

TempoClock also allows beat-based and bar-based scheduling, so it can be particularly useful when composing metric music. (See the TempoClock Help file for more details.)

Now let's take a look at Routines. A `Routine` is like a Function that you can evaluate a bit at a time, and in fact, you can use one almost anywhere you'd use a Function. Within a Routine, you use the yield method to return a value and pause execution. The next time you evaluate the Routine, it picks up where it left off.

```
(
r = Routine({
"foo".yield;
"bar".yield;
});
)
r.value; // foo
r.value; // bar
r.value; // we've reached the end, so it returns nil
```

Routine has a commonly used synonym for `value`, which is `next`. Although "next" might make more sense semantically with a Routine, "value" is sometimes preferable, for reasons we'll explore below.

Now here's the really interesting thing: since a Routine can take the place of a Function, if you evaluate a Routine in a Clock, and yield a number, the Routine will be rescheduled, just as in the SystemClock example above.

Figure 3.1 is a (slightly) more musical example that demonstrates a fermata of arbitrary length. This makes use of `wait`, a synonym for `yield`, and of Routine's `play` method, which is a shortcut for scheduling it in a clock. By yielding nil at a certain point, the clock doesn't reschedule, so you'll need to call `play` again when you want to continue, thus "releasing" the fermata. Functions understand a message called `fork`, which is a commonly used shortcut for creating a Routine and playing it in a Clock.

```
(
r = Routine({
   "foo".postln;
   1.yield; // reschedule after 1 second
   "bar".postln;
```

```
    1.yield;
    "foobar".postln;
});
SystemClock.sched(0, r);
)

// Fermata
s.boot;
(
r = Routine({
    x = Synth(\default, [freq: 76.midicps]);
    1.wait;

    x.release(0.1);
    y = Synth(\default, [freq: 73.midicps]);
    "Waiting. . .".postln;
    nil.yield;// fermata

    y.release(0.1);
    z = Synth(\default, [freq: 69.midicps]);
    2.wait;
    z.release;
});
)
// do this then wait for the fermata
r.play;
// feel the sweet tonic. . .
r.play;
```

Figure 3.1
A simple Routine illustrating a musical use of yield.

```
(
{
    "something".postln;
    1.wait;
    "something else".postln;
}.fork;
)
```

Figure 3.2 presents a similar example with a simple GUI control. This time, we'll use a `Task`, which you may remember from chapter 1. A Task works almost the same way that a Routine does, but it is meant to be played only with a Clock. A Task

provides some handy advantages, such as the ability to pause. Further, it prevents you from accidentally calling play twice. Try playing with the various buttons and see what happens.

```
(
t = Task({
   x = nil; // clear anything from previous code runs
   loop({   // loop the whole thing
        3.do({    // do this 3 times
              x.release(0.1); // release last Synth if needed
              x = Synth(\default, [freq: 76.midicps]);
              0.5.wait;
              x.release(0.1);
              x = Synth(\default, [freq: 73.midicps]);
              0.5.wait;
        });
        "I'm waiting for you to press resume".postln;
        nil.yield; // wait here for user to resume
        x.release(0.1);
        x = Synth(\default, [freq: 69.midicps]);
        1.wait;
        x.release;
   });
});

w = Window.new("Task Example", Rect(400, 400, 200, 30)).front;
w.layout = HLayout(
Button.new(w, Rect(0, 0, 100, 20)).states_([["Play/Resume",   Colo
r.black, Color.clear]])
   .action_({t.resume(0);}),
Button.new(w, Rect(0, 0, 40, 20)).states_([["Pause",   Color.black,
Color.clear]])
   .action_({t.pause;}),
Button.new(w, Rect(0, 0, 40, 20)).states_([["Stop",   Color.black,
 Color.clear]])
   .action_({
        t.stop;
        x.release(0.1);
        w.close;
   });
)
)
```

Figure 3.2
Using `Task` so you can pause the sequence.

Note that the example above demonstrates both fixed scheduling and waiting for a trigger to continue. The trigger needn't be from a GUI button; it can be almost anything, for instance, audio input. (See chapter 15.)

By combining all of these resources, you can control events in time in pretty complicated ways. You can nest Tasks and Routines or combine fixed scheduling with triggers; in short, anything you like. Figure 3.3 is an example that adds varying tempo to the mix, as well as adding some random events.

```
(
r = Routine({
   c = TempoClock.new; // make a TempoClock
   // start a 'wobbly' loop
   t = Task({
        loop({
              x.release(0.1);
              x = Synth(\default, [freq: 61.midicps, amp: 0.2]);
              0.2.wait;
              x.release(0.1);
              x = Synth(\default, [freq: 67.midicps, amp: 0.2]);
              rrand(0.075, 0.25).wait; // random wait from 0.1 to
0.25 seconds
        });
   }, c); // use the TempoClock to play this Task
   t.start;
   nil.yield;

   // now add some notes
   y = Synth(\default, [freq: 73.midicps, amp: 0.3]);
   nil.yield;
   y.release(0.1);
   y = Synth(\default, [freq: 79.midicps, amp: 0.3]);
   c.tempo = 2; // double time
   nil.yield;
   t.stop; y.release(1); x.release(0.1);
// stop the Task and Synths
});
)

r.next; // start loop
r.next; // first note
```

```
r.next; // second note; loop goes 'double time'
r.next; // stop loop and fade
```

Figure 3.3
Nesting tasks inside routines.

You can reset a Task or Routine by sending it the `reset` message:

```
r.reset;
```

3.2.2 Other Ways of Controlling Time in SC

There are two other notable methods of controlling sequences of events in SC: Patterns and the `Score` object. Patterns provide a high-level abstraction based on Streams of events and values. Since Patterns and Streams are discussed in chapter 6, we will not explore their workings in detail at this point, but it is worth saying that Patterns often provide a convenient way to produce a Stream of values (or other objects), and that they can be usefully combined with the methods shown above.

Figure 3.4 demonstrates two simple Patterns: `Pseq` and `Pxrand`. Pseq specifies an ordered sequence of objects (here, numbers used as durations of time between successive events) and a number of repetitions (in this case, an infinite number, indicated by the special value `inf`). Pxrand also has a list (used here as a collection of pitches), but instead of proceeding through it in order, a random element is selected each time. The "x" indicates that no individual value will be selected twice in a row; that is, it excludes the previous selection.

```
(
// random notes from lydian b7 scale
p = Pxrand([64, 66, 68, 70, 71, 73, 74, 76], inf).asStream;
// ordered sequence of durations
q = Pseq([1, 2, 0.5], inf).asStream;
t = Task({
   loop({
        x = Synth(\default, [freq: p.value.midicps]);
        q.value.wait;
        x.release(2);
   });
});
t.start;
)
t.stop; x.release(2);
```

Figure 3.4
Using patterns within a `Task`.

Patterns are like templates for producing Streams of values. In order to use a Pattern, it must be converted to a `Stream`, in this case using the `asStream` message. Once you have a Stream, you can get values from it by using the `next` or `value` message, just as with a Routine. (In fact, as you may have guessed, a Routine is a type of Stream as well.) Patterns are powerful because they are "reusable," and many Streams can be created from one Pattern template. (Chapter 6 will go into more detail regarding this.)

As an aside, and returning to the idea of flexibility, the value message above demonstrates an opportunity for polymorphism, which is a fancy way of saying that different objects understand the same message.[1] Since all objects understand "value" (most simply return themselves), you can substitute any object (a `Function`, a `Routine`, a number, etc.) that will return an appropriate value for `p` or `q` in the example above. Since `p` and `q` are evaluated each time through the loop, it's even possible to do this while the `Task` is playing. (See figure 3.5.) Taking advantage of polymorphism in ways like this can provide great flexibility.

```
(
p = 64; // a constant note
q = Pseq([1, 2, 0.5], inf).asStream; // ordered sequence   of durat
ions
t = Task({
   loop({
   x.release(2);
   x = Synth(\default, [freq: p.value.midicps]); // p may change
   q.value.wait;
   });
});
t.start;
)
// now change p
p = Pseq([64, 66, 68], inf).asStream; // to a Pattern: do re mi
p = {rrand(64, 76)};
// to a Function: random notes from a chromatic octave
t.stop; x.release(2);
```

Figure 3.5
Thanks to polymorphism, we can substitute objects that understand the same message.

The second method of controlling event sequences is the `Score` object. Score is essentially an ordered list of times and OSC commands. This takes the form of nested

Arrays. That is,

```
[
[time1, [cmd1]],
[time2, [cmd2]],
. . .
]
```

As you'll recall from chapter 2, OSC stands for Open Sound Control, which is the network protocol that SC uses for communicating between language and server. What you probably didn't realize is that it is possible to work with OSC messages directly, rather than through objects such as Synths. This is a rather large topic, and not the most common way of working, so since the OSC messages which the server understands are outlined in the Server Command Reference Help file, we'll just refer you there if you'd like to explore further. Figure 3.6 provides a short example. It is worth noting that the most common use for `Score` is for non-real-time synthesis (see chapter 18).

```
(
SynthDef("ScoreSine", {arg freq = 440;
Out.ar(0,
      SinOsc.ar(freq, 0, 0.2) * Line.kr(1, 0, 0.5, doneAction: 2)
)
}).add;
x = [
// s_new means create a new synth
// args for s_new are synthdef, nodeID, addAction, targetID, // syn
th args. . .
[0.0, [\s_new, \ScoreSine, 1000, 0, 0, \freq, 1413]],
[0.5, [\s_new, \ScoreSine, 1001, 0, 0, \freq, 712]],
[1.0, [\s_new, \ScoreSine, 1002, 0, 0, \freq, 417]],
[2.0, [\c_set, 0, 0]]
// dummy command to mark end of NRT synthesis time
];
z = Score(x);
)
z.play;
```

Figure 3.6
Using "messaging style" with `Score`.

3.2.3 Cue Players

Now let's turn to a more concrete example. Triggering sound files, a common technique when combining live performers with a fixed media part, is easily achieved in SuperCollider. There are many approaches to the construction of cue players, ranging from a list of individual lines of code that you evaluate one by one during a performance to fully fledged GUIs that completely hide the code from the user.

One question you need to consider is whether to play the sounds from RAM or stream them from hard disk. The latter may be more convenient for very long cues that you wouldn't want to keep in RAM. There are several classes (both in the standard distribution of SuperCollider and within extensions by third-party developers) that help with these 2 alternatives. Here's a very simple example which loads 2 files into RAM and plays them:

```
~myBuffer = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk0
1.wav"); // load an example sound
~myBuffer.play; // play it and notice it will release the node   //
after playing
```

Buffer's play method is really just a convenience method, though, and we'll probably want to do something fancier, such as fade in or out. Figure 3.7 presents an example which uses multiple cues in a particular order, played by executing the code one line at a time. It uses the `PlayBuf` UGen, which you may remember from chapter 1.

```
(
// here's a synthdef that allows us to play from a buffer, // with
 a fadeout
SynthDef("playbuf", {arg out = 0, buf, gate = 1;
   Out.ar(out,
        PlayBuf.ar(1, buf, BufRateScale.kr(buf), loop: 1.0)
              * Linen.kr(gate, doneAction: 2);
// release synth when fade done
   )
}).add;
// load all the paths in the example sounds/ folder into buffers
~someSounds = (Platform.resourceDir +/+"sounds/*").pathMatch   .col
lect {|path | Buffer.read(s, path)};
)
// now here's the score, so to speak
// execute these one line at a time
a = Synth("playbuf", [buf: ~someSounds[0]]);
a.release; b = Synth("playbuf", [buf: ~someSounds[1]]);
b.release; c = Synth("playbuf", [buf: ~someSounds[2]]);
```

```
c.release;
// free the buffer memory
~someSounds.do(_.free);
```

Figure 3.7
Executing one line at a time.

The middle two lines of the latter section of figure 3.7 consist of two statements, and thus they do two things when you press the enter key to execute. Of course, you can have lines of many statements, which can all be executed at once. (Lines are separated by carriage returns, statements by semicolons.) *Note:* This and figure 3.8 assume that all the files loaded are in mono. If stereo or greater, you may get a warning.

The "one line at a time" approach is good when developing something for yourself or an SC-savvy user, but you might instead want something a little more elaborate or user-friendly, for example when sending your piece to a nonspecialist. Figure 3.8 shows a similar example with a basic GUI.

```
(
SynthDef("playbuf", {arg out = 0, buf, gate = 1;
   Out.ar(out,
        PlayBuf.ar(1, buf, BufRateScale.kr(buf), loop: 1.0)
        * Linen.kr(gate, doneAction: 2) * 0.6;
        // with 'doneAction: 2' we release synth when fade is done
)}) .add;
~someSounds = (Platform.resourceDir +/+ "sounds/*").pathMatch   .co
llect {|path | Buffer.read(s, path)};
n = 0; // a counter
// here's our GUI code
w = Window.new("Simple CuePlayer", Rect(400, 400, 200, 30)).front;
w.layout = HLayout(
   //this will play each cue in turn
   Button.new(w, Rect(0, 0, 80, 20)).states_([["Play Cue",   Color.
black, Color.clear]]).action_({
        if(n < ~someSounds.size, {
             if(n != 0, {~nowPlaying.release;});
             ~nowPlaying = Synth("playbuf", [buf: ~someSounds
[n]]);   n=n+1;
        });
   }),
   //this sets the counter to the first cue
   Button.new(w, Rect(0, 0, 80, 20)).states_([["Stop / Reset",   Co
lor.black, Color.clear]]).action_({
```

```
            n=0;
            ~nowPlaying.release;
        })
    );
    // free the buffers when the window is closed
    w.onClose = {~someSounds.do(_.free);};
    )
```

Figure 3.8
Play cues with a simple GUI.

SC also allows for streaming files in from disk using the `DiskIn` and `VDiskIn` UGens (the latter allows variable-speed streaming). We refer you to their help files for further information.

As noted, the previous examples assume mono files. For multichannel files (stereo or more), it is simplest to deal with interleaved files.[2] In such cases, you must have a `SynthDef` for each required channel count (e.g., "playbuf-stereo"), with the number of channels specified in PlayBuf's first argument, as you can see in figure 3.9.

```
SynthDef("playbuf-stereo", {arg out = 0, buf, gate = 1;
    Out.ar(out,
            PlayBuf.ar(2, buf, BufRateScale.kr(buf), loop: 1.0)
            * Linen.kr(gate, doneAction: 2) * 0.6;
// channel count 2
)}) .add;
```

Figure 3.9
A `SynthDef` for stereo buffer playback.

3.3 Generating Sound Material

The process of composition deals as much with creating sounds as it does with ordering them. The ability to control sounds and audio processes at a low level can be great for finding your own compositional voice. Again, an exhaustive discussion of all of SuperCollider's sound-generating capabilities would far exceed the scope of this chapter, so we'll look at a few issues related to generating and capturing material in SC and give a concrete example of an approach you might want to adapt for your own purposes. As before, we will work here with sound files for the sake of convenience, but you should keep in mind that what we're discussing could apply to more or less any synthesis or processing technique.

3.3.1 Recording

At some point you're probably going to want to record SC's output for the purpose of capturing a sound for further audio processing or "assembly" on a DAW, documenting a performance, or converting an entire piece to a distributable sound file format.

To illustrate this, let's make a sound by creating an effect that responds in an idiosyncratic way to the amplitude of an input file and then record the result. You may not find a commercial plug-in that will do this, but in SC, you should be able to do what you can imagine (more or less!).

The `Server` class provides easy automated recording facilities in code. Often, this is the simplest way to capture your sounds precisely. (See figure 3.10.)

```
s.boot; // make sure the server is running
(  // first evaluate this section
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01   .wa
v"); // a source
s.prepareForRecord;
// prepare the server to record (you must do this first)
)
(   // simultaneously start the processing and recording
s.bind({
   // here's our funky effect
   x = {var columbia, amp;
        columbia = PlayBuf.ar(1, b, loop: 1);
        amp = Amplitude.ar(columbia, 0.5, 0.5, 4000, 250);   // 'st
icky' amp follower
        Out.ar(0, Resonz.ar(columbia, amp, 0.02, 3))
// filter; freq follows amp
        }.play;
s.record;
});
)
s.pauseRecording; // pause
s.record // start again
s.stopRecording;
// stop recording and close the resulting sound file
```

Figure 3.10
Recording the results of making sounds with SuperCollider.

After executing this code, you should have a sound file in SC's recordings folder (see the doc for platform-specific locations) labeled with the date and time that SC began recording: SC_YYMMDD_HHMMSS.aif.

If you are using the SuperCollider IDE, you can record server output just by starting recording under the Server Menu. In other editors, you may want to use the handy buttons on the Server window (execute `s.makeWindow`) to start and stop recording. On OSX it may look like this, or similar (see figure 3.11).

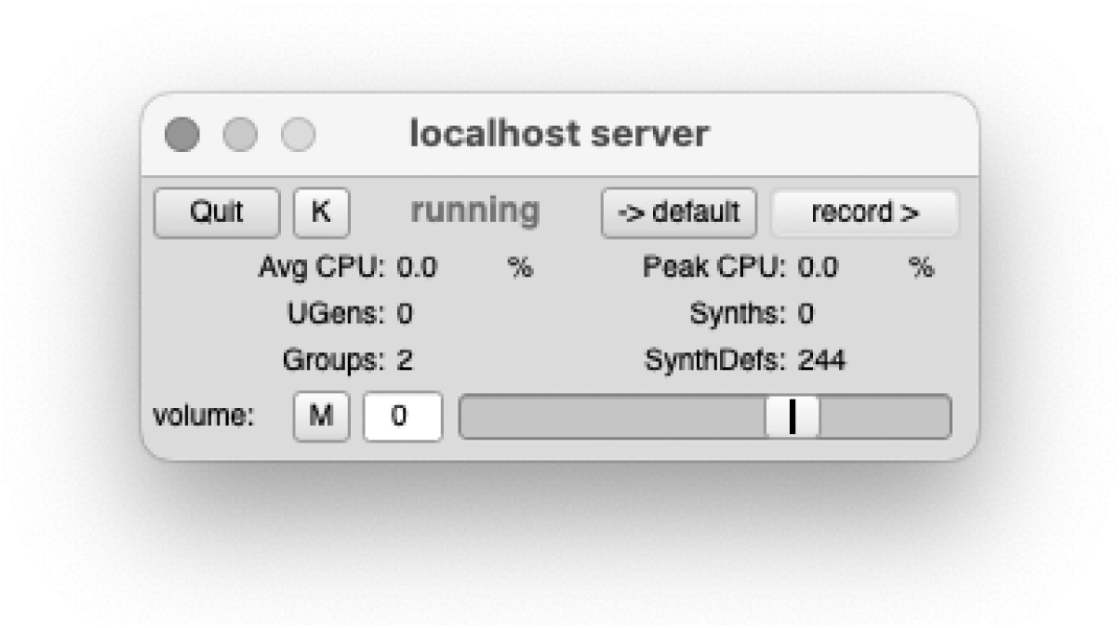


Figure 3.11
A Server Window

The above example uses the default recording options. Using the methods `prepareForRecord(path)`, `recChannels_`, `recHeaderFormat_`, and `recSampleFormat_`, you can customize the recording process. The latter three methods must be called before `prepareForRecord`. A common case is to change the sample format; the default is to record it as 32-bit floating-point values. This has the advantage of tremendous dynamic range, which means that you don't have to worry about clipping and can normalize later, but it's not compatible with all audio software:

```
s.recSampleFormat_("int16");
```

More elaborate recording can be realized by using the `DiskOut` UGen. Server's automatic functionality is in fact based on this. SC also has non-real-time synthesis capabilities, which may be useful for rendering CPU-intensive code. (See chapter 18.)

3.3.2 Thinking in the Abstract

Something that learners often find difficult to do is to stop thinking about exactly what they want to do at the moment and instead consider whether the problem they're dealing

with has a general solution. Generalizing your code can be very powerful. Imagine that we want to make a sound that consists of 3 bands of resonated impulses. We might do something like this:

```
(
{
Resonz.ar(Dust2.ar(5), 300, 0.001, 100) +
Resonz.ar(Dust2.ar(5), 600, 0.001, 100) +
Resonz.ar(Dust2.ar(5), 900, 0.001, 100) * 3.reciprocal;
// scale to ensure no clipping
}.play
)
```

Now, through a bit of careful thinking, we can abstract the problem from this concrete realization and come up with a more general solution:

```
(
f = 300;
n = 3;
{
Mix.fill(n, {|i| Resonz.ar(Dust2.ar(5), f * (i + 1), 0.001, 100)})
* n.reciprocal; // scale to ensure no clipping
}.play
)
```

This version has an equivalent result, but we've expressed it in terms of generalized instructions. It shows you how to construct a Synth consisting of resonated impulses tuned in whole-number ratios rather than as an exact arrangement of objects and connections, as you might do in a visual patching language such as Max/MSP. We've also used variables (`f` for frequency and `n` for number of resonators) to make our code easy to change. This is the great power of abstraction: by expressing something as a general solution, you can be much more flexible than if you think in terms of exact implementations. Now it happens that the example above is hardly shorter than the first, but look what we can do with it:

```
(
f = 40;
n = 50;
{
Mix.fill(n, {|i| Resonz.ar(Dust2.ar(5), f * (i + 1), 0.001, 300)})
* n.reciprocal; // scale to ensure no clipping
```

```
}.play
)
```

By changing `f` and `n` we're able to come up with a much more complex variant. Imagine what the hard-coded version would look like with 50 individual `Resonz` UGens typed out by hand. In this case, not only is the code more flexible, it's shorter; and because of that, it's much easier to understand. It's like the difference between saying "Make me 50 resonators" and saying "Make me a resonator. Make me a resonator. Make me a resonator …"

This way of thinking has potential applications in almost every aspect of SC, even GUI construction (see figure 3.12).

```
(
f = 100;
n = 30; // number of resonators
t = Array.fill(n, {|i|
{
Resonz.ar(Dust2.ar(5), f * (i + 1), 0.001, 300)
* n.reciprocal; // scale to ensure no clipping
}.play;
});

// now make a GUI
// a scrolling window so we don't run out of space
w = Window.new("Buttons", Rect(50, 100, 290, 250), scroll:true);
w.view.decorator = FlowLayout.new(w.view.bounds);
// auto layout the widgets
n.do({|i|
Button.new(w, Rect(0, 0, 130, 30)).states_([
["Freq" + (f * (i + 1)) + "On", Color.black, Color.white],
["Freq" + (f * (i + 1)) + "Off", Color.white, Color.black]
])
.action_({arg butt;
t[i].run(butt.value == 0);
});
});
w.front;
)
```

Figure 3.12
A variable number of resonators with an automatically created GUI

3.3.3 Gestures

For a long time, electroacoustic and electronic composition has been a rather "manual" process. This may account for the computer being used today as a virtual analog studio; many sequencer software GUIs attest to this way of thinking. However, as software has become more accessible, programming has somewhat replaced this virtual splicing approach.

One of the main advantages of a computer language is generalization, or abstraction, as we have seen above. In the traditional "taped" music studio approach, the composer does not differentiate gesture from musical content. In fact, traditionally they amount to much the same thing in electronic music. But can a musical gesture exist independently of sound?

In electronic music, gestures are, if you will, the morphology of the sound, a compendium of its behavior. Can we take sound material and examine it under another abstracted morphology? In ordinary musical terms this could mean a minor scale can be played in crescendo or diminuendo and remain a minor scale. In electroacoustic music, this can happen, for example, when we modulate one sound with the spectrum of another. The shape of one sound is generalized and applied to another; we are accustomed to hearing this in signal-processing software. In this section, we would like to show how SuperCollider can be used to create "empty gestures," gestures that are not linked to any sound in particular. They are, in a sense, gestures waiting for a sound, abstractions of "how to deliver" the musical idea.

First we will look at some snippets of code that we can reuse in different patches, and then we will look at some Routines we can call up as part of a "Routine of Routines" (i.e., a score, so to speak). If you prefer to work in a more traditional way, you can just run the Routines with different sounds each time, record them to hard disk, and then assemble or sample as usual in your preferred audio editing/sequencing software. However, an advantage of doing the larger-scale organization of your piece within SC is that since you are interpreting your code during the actual performance of your piece, you can add elements of variability to what is normally fixed at the time of playback. You can also add elements of chance to your piece without necessarily venturing fully into algorithmic composition. (Naturally, you can always record the output to a sound file if desired.) This, of course, brings us back to issues of design, and exactly what you choose to do will depend on your own needs and inclinations.

3.3.4 Making "Empty" Gestures

Let's start by making a list where all our Buffers will be stored. This will come in handy later, as it will allow us to call up any file we opened with our file browser during the course of our session. In the following example, we open a dialog box and can select any sounds on our hard disk:

```
(
//you will be able to add multiple sound files; just shift   // cli
ck when selecting!
var file, soundPath;
~buffers = List[];
Dialog.openPanel({arg paths;
paths.do({|soundPath|
//post the path to verify that it is the one you expect!
  soundPath.postln;
//adds the recently selected Buffer to your list
  ~buffers.add(Buffer.read(s, soundPath););})
}, multipleSelection: true);
)
```

You can check to see how many Buffers are in your list so far (watch the post window!):

```
~buffers.size;
```

and you can see where each sound is inside your list. For example, here is the very first sound stored in our Buffer list:

```
~buffers[0];
```

Now that we have our sound in a Buffer, let's try some basic manipulations. First, let's just listen to the sound to verify that it is there:

```
~buffers[0].play;
```

Now, let's make a simple `SynthDef` so we can create Synths which play our Buffer (for example, in any `Routine`, `Task`, or other `Stream`) later on. For the purposes of this demonstration, we will use a very simple percussive envelope, making sure we have `doneAction: 2` in order to free the synth after the envelope terminates:

```
(
// buffer player with done action and control of envelope and // pa
nning
SynthDef(\samplePlayer, {arg out = 0, buf = 0,
  rate = 1, at = 0.01, rel = 0.1, pos = 0, pSpeed = 0, lev = 0.5;
  var sample, panT, amp, aux;
  sample = PlayBuf.ar(1, buf, rate*BufRateScale.kr(buf), 1, 0, 0);
```

```
    panT= FSinOsc.kr(pSpeed);
    amp = EnvGen.ar(Env.perc(at, rel, lev), doneAction: 2);
    Out.ar(out, Pan2.ar(sample, panT, amp));
}).add;
)
```

As mentioned in chapter 1, we use the `add` method here rather than one of the lower-level SynthDef methods such as `send`. In addition to sending the `def` to the server, `add` also stores it within the global `SynthDescLib` in the client app, so that its arguments can be looked up later by the Patterns and Streams system (see chapter 6). We'll need this below. Let's test the SynthDef:

```
Synth(\samplePlayer,[\out, 0, \buf, ~buffers[0], \rel, 0.25]);
```

As you can hear, it plays 0.25 second of the selected sound. Of course, if you have made more than one Buffer list, you can play sounds from any list, and also play randomly from that list. For example, from the list we defined earlier we could do this:

```
Synth(\samplePlayer,[\out, 0, \buf, ~buffers.choose, \rel, 0.25]);
```

Let's define a Routine that allows us to create a stuttering/rushing gesture in a glitch style. We'll use a new Pattern here, `Pgeom`, which specifies a geometric series.[3] Note that Patterns can be nested. Figure 3.13 shows a `Pseq` whose list consists of two `Pgeoms`.

```
(
/* a routine for creating a ritardando stutter with panning,   you
 must have run the code so that this routine may find   some sounds
already loaded into buffers, you can change the   index of ~buffere
dCues to test the routine on different sounds */

~stut = Routine ({var dur, pos;
~stutPatt = Pseq([Pgeom(0.01, 1.1707, 18), Pn(0.1, 1),   Pgeom(0.1,
0.94, 200)]);
~str= ~stutPatt.asStream;
100.do{
dur = ~str.next;
dur.postln;    //so we can check values on the post window
~sample = Synth("samplePlayer",[\out, 0, \buf, ~buffers[0],   \at,
 0.1, \rel, 0.05,\pSpeed, 0.5]);
dur.wait;
```

```
}
});
)

//now play it
~stut.play;
// reset before you play again!
~stut.reset;
```

Figure 3.13
Making a stuttering gesture using a geometric pattern.

Remember that you can use a `Task` or `Routine` to sequence several such gestures within your piece. You can, of course, modify the Routine to create other accel/decel Patterns by substituting different Patterns. You can also add variability by making some of them perform choices when they generate their values (e.g., using `Prand` or `Pxrand`). You can use this, for example, to choose which speaker a sound comes from without repeating speakers:

```
Pxrand([0, 1, 2, 3, 4, 5, 6, 7], inf)
```

The advantage of having assigned your gestures to environment variables (using the tilde shortcut) is that now you are able to experiment in real time with the ordering, simultaneity, and internal behavior of your gestures.

Let's take a quick look at one more important Pattern: `Pbind`. It creates a Stream of Events, which are like a kind of dictionary of named properties and associated values. If you send the message `play` to a Pbind, it will play the Stream of Events, in a fashion similar to the Clock examples above. Here's a simple example which makes sound using what's called the "default" SynthDef:

```
// randomly selected frequency, duration 0.1 second
Pbind(\freq, Prand([300, 500, 231.2, 399.2], 30), \dur, 0.1).   pla
y;
```

It's also possible to substitute Event Streams as they play. When you call `play` on a `Pattern`, it returns an `EventStreamPlayer`, which creates the individual Events from the Stream defined by the Pattern. EventStreamPlayer allows its Stream to be substituted while it is playing.

```
~gest1 = Pbind(\instrument, \samplePlayer, \dur, 2, \rel, 1.9);
~player = ~gest1.play; //make it play
```

```
~player.stream = Pbind(\instrument, \samplePlayer, \dur, 1/8,   \ra
te, Pxrand([1/2, 1, 2/3, 4], inf), \rel, 0.9).asStream;   //substit
ute the stream
~player.stop;
```

If you have evaluated the expressions above, you will notice that you don't hear the simple default SynthDef, but rather the one we made earlier. Since we added it above, the Pbind is able to look it up in the global library and get the information it needs about the def. Now, the Pbind plays repeatedly at intervals specified by the \dur argument, but it will stop playing as soon as it receives nil for this or any other argument. So we can take advantage of this to make Streams that are not repetitive and thus make single gestures (of course, we can also choose to work in a looping/layering fashion, but more of that later). Here is a Pbind making use of our accelerando Pattern to create a rushing sound:

```
~gest1 = Pbind(\instrument, \samplePlayer, \dur, Pgeom(0.01,   1.17
07, 20), \rel, 1.9);
~gest1.play;
```

When the Stream created from the Pgeom ended, it returned nil and the EventStreamPlayer stopped playing. If you call play on it again, you will notice that it makes the same rushing sound without the need to reset it, as we had to do with the Routine, since it will return a new EventStreamPlayer each time. More complex gestures can be made, of course, by nesting patterns:

```
Pbind(\instrument, \samplePlayer, \dur, Pseq([Pgeom(0.01,   1.1707,
20), Pgeom(0.01, 0.93, 20)], 1), \rel, 1.9,   \pSpeed, 0.5).play;
Pbind(\instrument, \samplePlayer, \dur, Pseq([Pgeom(0.01,   1.1707,
20), Pgeom(0.01, 0.93, 20)], 1),\rate, Pxrand([1/2,   1, 2/3, 4], i
nf), \rel, 1.9, \pSpeed, 0.5).play;
```

Similar things can be done with the `Pdef` class from the JIT library (see chapter 7). Let's designate another environment variable to hold a sequence of values that we can plug in at will and change on the fly. This Pattern holds values that would work well for `\dur`:

```
~rhythm1 = Pseq([1/4, 1/4, 1/8, 1/12, 1/24, nil]); //the nil   is s
o it will stop!
```

We can then plug it into a `Pdef`, which we'll call `\a`:

```
Pdef(\a, Pbind(\instrument, \samplePlayer, \dur, ~rhythm1,   \rel,
 1.9, \pSpeed, 0.5)) .play;
```

If we define another sequence of values we want to try,

```
~rhythm1 = Pseq([1/64, 1/64, 1/64, 1/32, 1/32, 1/32, 1/32,   1/24,
 1/16, 1/12, nil]);
```

and then reevaluate the Pdef,

```
Pdef(\a, Pbind(\instrument, \samplePlayer, \dur, ~rhythm1,   \rel,
 1.9, \pSpeed, 0.5)); // already playing,
```

we can hear that the new `~rhythm1` has taken the place of the previous one. Notice that it played immediately, without the need for calling `play`. This is one of the advantages of working with the `Pdef` class: once the `Stream` is running, anything that is "poured" into it will come out. In the following example, we assign a Pattern to the rate values and obtain an interesting variation:

```
Pdef (\a, Pbind(\instrument, \samplePlayer, \at, 0.5, \rel, 3,   \l
ev, {rrand(0.1, 0.2)}, \dur, 0.05, \rate, Pseq([Pbrown(0.8,   1.01,
0.01, 20)])));
```

Experiments like these can be conducted by creating Patterns for any of the arguments that our SynthDef will take. If we have "added" more than one SynthDef, we can even modulate the `\instrument` by getting it to choose among several different options. Once we have a set of gestures that we like, we can trigger them in a certain order using a `Routine`, or we can record them separately and load them as audio files to our audio editor. The latter approach is useful if we want to use a cue player for the final structuring of a piece.

3.4 Conclusions

What next? The best way to compose with the SuperCollider is to set yourself a project with a deadline. In this way, you will come to grips with specific things you need to know, and you will learn it much better than just by reviewing everything it can do. SuperCollider offers a variety of approaches to electronic music composition. It can be used for sound creation thanks to its rich offering of UGens (see chapter 2), as well as for assembling your piece in flexible ways. We have shown that the assembly of sounds itself can become a form of synthesis, illustrated by our use of Patterns and Streams.

Another approach is to review some of the classic techniques used in electroacoustic composition and try to recreate them yourself using the SuperCollider. Below we refer you to some interesting texts that may enhance your creative investigations.

Notes

1. You may have noticed that the terms "message" and "method" used somewhat interchangeably. In polymorphism, the distinction becomes clear: different objects may respond to the same message with different methods. In other words, the message is the command, and the method is what the object does in response.
2. Scott Wilson's De-Interleaver application for OSX and Jeremy Friesner's cross-platform command line tools audio_combine and audio_split allow for the convenient interleaving and deinterleaving of audio files.
3. A geometric series is a series with a constant ratio between successive terms.

References

Budón, O. 2000. "Composing with Objects, Networks, and Time Scales: An Interview with Horacio Vaggione." *Computer Music Journal,* 24(3): 9–22.

Collins, N. 2010. *Introduction to Computer Music*. Chichester, UK: Wiley.

Dodge, C., and T. A. Jerse. 1997. *Computer Music: Synthesis, Composition, and Performance,* 2nd ed. New York: Schirmer.

Holtzman, S. R. 1981. "Using Generative Grammars for Music Composition." *Computer Music Journal,* 5(1): 51–64.

Loy, G. 1989. "Composing with Computers: A Survey of Some Compositional Formalisms and Music Programming Languages." In *Current Directions in Computer Music Research,* ed. M. V. Mathews and J. R. Pierce, pp. 291–396. Cambridge, MA: MIT Press.

Loy, G., and C. Abbott. 1985. "Programming Languages for Computer Music Synthesis, Performance, and Composition." *ACM Computing Surveys (CSUR)*, 17(2): 235–265.

Mathews, M. V. 1963. "The Digital Computer as a Musical Instrument." *Science,* 142(3592): 553–557.

Miranda, E. R. 2001. *Composing Music with Computers*. London: Focal Press.

Roads, C. 2001. *Microsound.* Cambridge, MA: MIT Press.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

Wishart, T. 1994. *Audible Design: A Plain and Easy Introduction to Practical Sound Composition*. York, UK: Orpheus the Pantomime.

4 Ins and Outs: SuperCollider and External Devices

Marije A. J. Baalman, Miguel Negrão, Stefan Kersten, and Till Bovermann

4.1 Introduction

Sometimes SuperCollider alone isn't enough, and you want to have input from external devices or other programs, to use output in other programs, or to control devices from SC. Common examples are interfacing with programs such as Processing[1] (Reas and Fry, 2007), and PureData[2] (Puckette, 2007). Devices can be anything ranging from MIDI keyboards, to game devices, to custom (self-)built sensor interfaces. This chapter will provide an overview of the most common ways of getting data into and out of SC. Because of limited space we will not provide a detailed explanation of the devices or protocols themselves, nor in most cases of any associated technical terminology. We suggest that you research these using the Internet and/or standard texts on computer music or computer hardware if you need more information. Where appropriate, however, we will provide references to Help files, Quarks (the name for certain SuperCollider extensions which can be pulled in and updated via commands from the language; see the Quarks Help file), external resources, and other chapters in this book.

SuperCollider provides many ways to work with external data and to send external data. In this chapter, we will look into Open Sound Control, MIDI, HIDs, SerialPort, analog sensing, and making custom interfaces with Bela. Another option to interface with an external program would be by piping data (see the `Pipe` Help file). Toward the end of the chapter, we look at the Modality Toolkit, which provides a cross-protocol way of getting data from and to devices, and provides GUI representations of devices.

Let's say you have a device that you want to use as a musical interface to perform your music, or you may have an idea for an interface and be looking for the right way to build it. In either case the questions you will have are the following:

1. How do I get data from the device?
2. How do I send data to the device?
3. How do I create a sensible mapping?

The first two questions are mostly of a technical nature and will be described in detail in the next section for different protocols; the last question is quite complicated

and is tightly linked to your artistic purposes. We will present some basic approaches to get started with tackling this question. We will not go into the mechanical questions of what sensors to use to capture the parameters that one wants to measure in the real world, as this goes beyond the scope of this chapter.

4.2 Getting the Data

4.2.1 Human Interface Devices

Human interface devices (HID) are devices through which the external world can communicate with computers. Most common are the keyboard and the mouse; other examples are gamepads and joysticks. These devices are nowadays usually connected via USB (Universal Serial Bus).

For USB an HID specification has been defined. This ensures that computers running different operating systems can recognize a connected device as an HID and load the appropriate driver for it so that programs can use its data. Although HIDs are mostly used for input, it is sometimes also possible to send output to them. The latter is usually done to give the user feedback, for example, the Caps Lock LED that is found on many keyboards.

4.2.1.1 HID, HIDFunc, and HIDdef HID provides a cross-platform interface to access HIDs. The basic approach is the following:

1. Find available devices: `HID.findAvailable;`
2. Print a readable list of available devices: `HID.postAvailable;`
3. Open a specific HID: `~myhid = HID.open (1103, 53251);`
4. Check whether data is coming in: `HID.debug = true;` or `HIDFunc.trace (true);`
5. Assign functions to be executed based on incoming data. We will elaborate on this below.

The method `findAvailable` will check which devices are connected to the computer and retrieve basic information about them. With the method `postAvailable` the list of devices is then printed to the post window in a readable list, in the first column the index into the device list is given, in the second column various properties of the device are listed. These properties can be used to open a specific device; you can either open a device with its index in the device list (`HID.openAt(0)`), by using the path in the operating system (`HID.openPath ("/dev/hidraw4")`), or by using the vendor ID and product ID (`HID.open(1103, 53251)`). You can check which devices have already been opened by SuperCollider with `HID.openDevices`.

Once you have opened an HID, you can check its capabilities with `~myhid.postElements`, `~myhid.postInputElements`, and

`~myhid.postOutputElements`. These will print the properties of each element. You can also use `~myhid.postUsages`, which will sort the elements by their usage.

For each element, the information in the post window will look like this:

```
HID Element: 18, type: 1, 2, usage page: 1, usage index: 53
   Description: GenericDesktop, Rz, input,
    [Data, Variable, Absolute, NoWrap, Linear, PreferredState, NoNu
llPosition, NonVolatile, BitField]
   Usage range: [—1, —1]
   Logical range: [0, 255]
   Physical range: [0, 255], Unit: 0, Exponent: 0
   Report ID: 0, size 8, index 5
```

The type is a numerical index of whether it is an input, output, or feature element; the second one indicates other properties. In the description, the page and usage are translated using the table lookup, and the type indication is also translated to something understandable. The usage range is again important for keyboards (as one HID element will report the state of several keys at once), the logical and physical ranges have various values. The report ID, size, and index give low-level information on how the data come in.

You can then assign an action to be performed when data are received at different levels: the global level (for any HID device, with `HID.action`), the device level (with `~myhid.action`), or the element level (`~myhid.elements.at(18).action = {arg value, element; "HID element action:".post; [value, element].postln;};`). The classes `HIDFunc` and `HIDdef` provide an interface for assigning actions that is similar to the interfaces you'll see below for interfacing with MIDI or OSC. The advantage of these two functions is that they are independent of the actual HID device that is opened, but rather allow you to filter based on a usage type of an element or of a device. So you can define functions that will work for any opened gamepad, for example. This makes it easy to use the same code with different physical devices or to distribute your code to other users, who may have a gamepad, but not necessarily the exact same type that you do.

With `HIDFunc`, you can filter with the methods.`usage`, `.device`,.`usageID`, `.proto`, and `.element`. The recommended method is `HIDFunc.usage`. For example, to filter all events coming from the *x*-axis of a mouse, use the following:

```
a = HIDFunc.usage({|. . .args| args.postln;}, \X, \Mouse);
```

And to disable the function again, use `a.free;`

`HIDdef` provides an interface where you do not need to assign the function to a variable, but rather give the function a label. This makes it easier to live-code or reexecute the same line (or altered line) again and again and replace the original definition. In the following code snippet, we give the label `\example` to the `HIDdef`, but otherwise perform the same function:

```
HIDdef.usage(\example, {|. . .args| args.postln;}, \X, \Mouse);
```

A more elaborate example is given in the website materials for this chapter in the file gamepad_example.scd.

4.2.1.2 Server-side methods of accessing HID devices For some input devices, there are specific UGens that access their data (for example, the mouse UGens `MouseX`, `MouseY`, and `MouseButton`, and the keyboard UGen `KeyState`, all of which are included in SuperCollider). There are several others available as extensions. The advantage of accessing data server-side is that you do not need to pass on data from the language to the server in order to affect a synthesis process. The disadvantage is that there is less flexibility in the mapping of the data. An example is shown in figure 4.1.

```
(
SynthDef(\mouseExample, {
   Out.ar(0,
        Resonz.ar(WhiteNoise.ar,
              MouseX.kr(400, 3000,\exponential), // mouse x is   fr
equency
              MouseY.kr(0.01, 2, \exponential) // mouse y is   band
width
        ) * MouseButton.kr // mouse button is on and off
   );
}).add;
);
// create an instance:
x = Synth(\mouseExample);
// now move the mouse to different places on the screen and push th
e left button. If you keep it pressed and move the mouse, you sweep
through the filter.
// free the Synth again:
x.free;
```

Figure 4.1
Example of using the `Mouse` UGens.

4.2.1.3 GUI methods of accessing HID devices

SuperCollider GUI widgets provide the ability to capture keyboard and mouse events through callback functions such as `mouseDownAction` and `keyDownAction`. Since these are discussed in detail in chapter 12, they will not be explored in any depth here. A simple example is presented in figure 4.2.

```
(
w = Window.new("Key Example", Rect(0,0, 150, 150));
c = UserView(w, Rect(0, 0, 150, 150)).background_(Color.white);
c.keyDownAction = {arg view, char, modifiers, unicode, keycode;
  [char, modifiers, unicode, keycode].postln;
};
w.front;
);
```

Figure 4.2
Example of capturing keyboard events using the GUI.

From an interaction design viewpoint, using a GUI widget to capture an HID event should be done only if the interaction only makes sense with that GUI widget (e.g., if the GUI widget provides visual feedback that is essential to the interaction).

4.2.2 MIDI

MIDI (Musical Instrument Digital Interface)[3] is an asynchronous serial protocol that was introduced in 1983. A MIDI message consists of 3 bytes; the first is a status byte containing the message type and channel, and the other two are data bytes.

Devices that send or receive MIDI are typically keyboards, hardware synthesizers, drum pads, and so on, but there are also several sensor interfaces available which use MIDI.

In the following we will look at how to initialize access to a MIDI device, how to get input from it, and how to send output to it.

4.2.2.1 Initializing the MIDI connection

To initialize the MIDI connection, you can execute the code: `MIDIClient.init`. This connects SuperCollider to the MIDI subsystem of the operating system and checks which MIDI sources (devices or programs that generate MIDI) and MIDI destinations (devices or programs that can receive MIDI) are available. By default, the function opens as many MIDI ports as it finds sources and destinations. You can also specify how many input ports and output ports to create for SuperCollider by passing them as arguments; for example, `MIDClient.init(1,2)` will create 1 input port and 2 output ports.

Each device or program that is found with the `MIDIClient.init` method will become available as a `MIDIEndpoint` in the lists `MIDIClient.sources` and `MIDIClient.destinations`.[4]

If at any point you want to disconnect SuperCollider from the MIDI subsystem, you can use `MIDIClient.disposeClient` or restart the connection with `MIDIClient.restart`.

4.2.2.2 MIDI input After initialization, you can connect to a single MIDI device to receive input by using `MIDIIn.connect`, or just connect to all MIDI input devices by using `MIDIIn.connectAll`.

To receive input from a MIDI device, you can use the classes `MIDIFunc` and `MIDIdef`.[5] These allow one to register functions that will be evaluated based on incoming messages. These are similar to `HIDFunc` and `HIDdef` (see above).

To just see what is coming in from any MIDI device that is connected, execute the following code:

```
MIDIIn.connectAll; // connect to all MIDI devices
MIDIFunc.trace(true); // trace the incoming MIDI messages
```

When you then move a knob or press a key or a pad on your MIDI device, you will then see messages like

```
MIDI Message Received:
   type: noteOn
   src: 1572864
   chan: 0
   num: 39
   val: 127
MIDI Message Received:
   type: noteOff
   src: 1572864
   chan: 0
   num: 39
   val: 127
MIDI Message Received:
   type: control
   src: 1572864
   chan: 0
   num: 1
   val: 71
```

In this code, `src` is the MIDI device that is sending the data (you can compare the number with the output of `MIDIClient.sources`); the other values give information on the type of message, the MIDI channel it was sent on, the note or control number, and the velocity or the control value. You can use each of these parameters to filter a MIDI message with `MIDIFunc`.

For example, to listen to `noteOn` messages, we can run

```
n = MIDIFunc.noteOn({|. . . args| args.postln});
```

This will post the velocity, note number, channel and source for any MIDI `noteOn` message:

```
[108, 39, 0, 1572864]
```

If we want to filter for a particular note, we can run

```
m = MIDIFunc.noteOn({|velocity| velocity.postln}, 39);
```

Now, if we play note number 39, we will get a post with the velocity (from `MIDIFunc m`), and of the array of arguments (from `n`). Similarly, you can filter for a range of note numbers (by passing in an array of notes, like `[39,40,41]`), the MIDI channel (3rd argument), the source (4th), and even a selection of velocities, for example to only react to note-on messages with a high velocity:

```
l = MIDIFunc.noteOn({|velocity| "!!! ".post; velocity.   postln}, 3
9, argTemplate: {|v| v>100});
```

The `MIDIFunc`s are automatically removed when all processes are stopped (default shortcut [Cmd/Ctrl]+[.]). You can make the `MIDIFunc` permanent with: `n.fix` or `n.permanent`. Or you can manually remove them with `n.remove`.

In table 4.1, we list the different methods of `MIDIFunc`, their counterparts for the `MIDIIn` class, and descriptions of what they do.

Table 4.1
MIDI messages and their corresponding implementations in SuperCollider

MIDIFunc Method	MIDIn Method	Description
Main Messages		
noteOn	noteOn	Note on
noteOff	noteOff	Note off

MIDIFunc Method	MIDIn Method	Description
cc	control	Control message
bend	bend	Pitch bend
touch	touch	After touch
polytouch	polytouch	Polyphonic aftertouch/pressure
program	program	Program change
System Messages		
sysex	sysex	System exclusive
sysrt	sysrt	Song select, clock, and transport control
smpte	smpte	MIDI time code
midiClock	–	MIDI clock
mtcQuarterFrame	–	MIDI time code quarter frame
tick	–	MIDI tick
start	–	Start playing
stop	–	Stop playing
continue	–	Continue playing
songPosition	–	Song position
songSelect	–	Song select
tuneRequest	–	Tune request
activeSense	–	Active sense
reset	–	Reset

One note about the MIDI note off message: not all programs and devices implement this message; some may use a MIDI note on message with a velocity of 0 instead.

A more elaborate example is given in the website materials for this chapter in the file *midi_example.scd*.

4.2.2.3 MIDI output In order to send MIDI output, an instance of `MIDIOut` needs to be created. `MIDIOut` takes two parameters: the output port number and the device to which it sends output (on Linux, this parameter is optional, but you need to additionally connect to the MIDI device using `MIDIOut.connect`). Then, from this instance, you can send any MIDI message: `noteOn`, `noteOff`, `polyTouch`, `control`, `touch`, `bend`, `allNotesOff`, `smpte`, and so on. The first argument to these messages is the MIDI channel, the second the note number, control number, or similar, and the third value the velocity, control value, or similar.

In figure 4.3 is an example that sends out MIDI using a `Task`.

```
MIDIClient.init;
m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
(
t = Task({
```

```
    [60, 64, 61, 60, 65, 61].do{|it|
         m.noteOn(16, it, 60); // channel 16 will send to all   chan
nels
         1.0.wait;
         m.noteOff(16, it, 60);
    };
    [64, 65, 67].dup(4).flatten.do{|it|
         m.noteOn(16, it, 120);
         0.25.wait;
         m.noteOff(16, it, 120);
    };
    [61, 65, 60, 61, 64, 60].do{|it|
         m.noteOn(16, it, 60);
         1.0.wait;
         m.noteOff(16, it, 60);
    };
    m.noteOn(16, 60, 40); m.noteOn(16, 52, 40);
    2.0.wait;
    m.noteOff(16, 60, 40); m.noteOff(16, 52, 40);
    m.noteOn(16, 57, 40); m.noteOn(16, 52, 40);
    2.0.wait;
    m.noteOff(16, 57, 40); m.noteOff(16, 52, 40);
    m.noteOn(16, 53, 40);
    2.0.wait;
    m.noteOff(16, 53, 40);
    m.noteOn(16, 52, 40);
    4.0.wait;
    m.noteOff(16, 52, 40);
});
);
t.play; // start playing the task
t.stop; // stop playing the task
```

Figure 4.3
Example of sending out MIDI using a `Task`.

See also
Help files: MIDI, UsingMIDI, MIDIFunc, MIDIdef, MIDIOut, MIDIIn.
Quarks: ddwMIDI

4.2.3 Open Sound Control

Open Sound Control (OSC) is a protocol designed at Berkeley's CNMAT for exchanging control data between audio applications[6] (Wright et al., 2003), but it is general enough to allow for a wide range of possible applications. OSC itself is

transport independent; that is, the underlying transport protocol is—except for the details of opening and closing connections—of little interest to higher application layers. OSC has been used with transports ranging from serial communication to high-speed Ethernet communication, but the most popular protocols are those of the IP family, because of their widespread support in operating systems.

4.2.3.1 UDP and TCP addressing and ports Before we delve into the details of sending and receiving OSC messages in SuperCollider, let's take a short look at the two most important IPs, User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). Although some details differ, both protocols have a common way of addressing services on a particular machine (which might be the same as the sending one). Computers (hosts) within the same network segment are uniquely identified by their IP address, a 32-bit (in IPv4) or 64-bit (in IPv6) number. IPv4 addresses are usually given in "dot" notation (i.e., by four numbers of 8 bits each, separated by dots: 192.168.1.2).

The special address 127.0.0.1 usually denotes the local host address (i.e., the address of the machine that the sending or receiving program is running on. Numeric IP addresses are hard to memorize, and in many networks, a host providing Domain Name Service (DNS) makes it possible to resolve host addresses based on a hierarchical naming scheme (host.domain.suffix). Each host is assigned a unique name within the network which can be used instead of the numeric IP address (e.g., supercollider.github.io). The special host name localhost is usually mapped to the IP address 127.0.0.1, and there is a similar mechanism for link-local addresses—used, for example, by Apple in its Rendezvous implementation—the domain .local (note the missing suffix).

In order to differentiate among different services on the same host, each program allocates a resource (a socket) which is bound to a specific integer port. Usually, ports below 1024 are reserved for system services; user applications are free to use any port number above 1024. The default port for *scsynth* is 57110, and the one for *sclang* and *SuperCollider.app* is 57120.[7] The combination of IP address and port makes it possible to address any service on any machine seen by the sending host.

The UDP is message oriented (i.e., communication takes place by sending individual packets of limited size, without any guarantee of correct delivery). It is used in many audio and video applications because of its lower overhead compared with TCP and its ease of use. It is also the default protocol used by *sclang* and *scsynth*. TCP, on the other hand, is connection oriented (i.e., conceptually, communication is an exchange of infinite Streams of unstructured binary data between two connected hosts). Since OSC itself is message based, the Stream needs to be "packetized" by prepending a 32-bit byte count to each OSC packet before sending. (Note that the SC objects discussed below will do this for you automatically.)

4.2.3.2 OSC messages in SuperCollider Among other things, as noted in chapter 3, SuperCollider uses OSC for controlling the synthesis server *scsynth* from the language application, so let's look at how the concepts explained above are realized in *sclang*. Hosts and services are represented by objects of the type `NetAddr`.

```
// Create a network address representing sclang itself
~host = NetAddr("localhost," NetAddr.langPort);
```

OSC packets come in 2 varieties: messages and bundles. Messages represent individual commands (e.g., triggering an action in the receiving host). A command is denoted by an ASCII string that is in a hierarchical format similar to Unix file system paths (e.g., `"/synth/filter"`). Bundles are collections of messages paired with a 64-bit time tag that denotes the intended time of execution in the standardized NTP format. Messages in the same bundle are guaranteed to be executed *atomically* by the receiving host. (That is, either they are all executed at the same logical time or none of them is executed. Logical time represents the scheduled time at which the events are executed and is distinct from physical time, which of course still advances while the execution occurs.) Being able to schedule messages slightly ahead of time is particularly important in audio synthesis in order to rectify the messaging jitter introduced by system-level process scheduling and network transport.

OSC messages are represented in SuperCollider by Arrays containing the command string and any number of arguments of primitive types (e.g., `String`, `Number`, `Boolean`, `Nil`, etc.) and Arrays encoded as binary "blobs." The method `sendMsg` of `NetAddr` takes a list of arguments and sends them as an OSC message:

```
// Send an OSC message
~host.sendMsg("/testMsg", 42, "string", pi);
```

To send a bundle, we use the `sendBundle` method of `NetAddr`, providing a time tag as an offset from the current time and a list of messages:

```
// Send an OSC bundle and execute its contents
// 200 ms from "now"
~host.sendBundle(0.2,
  ["/testMsg", 42, "string", pi],
  ["/testMsg", 183]);
```

Until now, we've sent messages to the language without actually reacting to them; in order to have a piece of code executed whenever a particular OSC message is received,

we use instances of the `OSCFunc` class or its counterpart `OSCdef` (similar again to `HIDFunc`, `MIDIFunc` and `HIDdef` and `MIDIdef`):

```
// Create an OSCFunc
(
~osc = OSCFunc({|msg, time, addr, recvPort|
   [msg, time, addr, recvPort].postln;
},"/testMsg", ~host)
);
```

This example creates an `OSCFunc`, so that the function specified will be executed whenever the message `"/testMsg"` is received from the network address `~host`. If a bundle containing multiple `"/testMsg"` messages is received, the logical time at which the registered function will be executed will be the same for all messages in the bundle.

Sometimes the address of the sending host is unknown, and therefore it's possible to leave out the argument for the `srcID` argument, or pass `nil`; an `OSCFunc` created like this will execute its action whenever a specific command is received from *any* address.

If the application that is sending OSC can only send to a particular port, you can open that port with SuperCollider by defining the argument `recvPort` of an `OSCFunc`. If you do, SuperCollider will attempt to open the given port, and the `OSCFunc` will only react to messages send to that port. You can always check which ports SuperCollider has opened with: `thisProcess.openPorts`.

As with to MIDI and HID, you can use the method `OSCFunc.trace(true);` to post any incoming OSC messages. When *scsynth* is running, you will see regular status messages arrive from the server; you can suppress these by passing in a second argument (`hideStatusMsg`): `OSCFunc.trace(true, true);`

SuperCollider's implementation of the OSC specification is idiosyncratic in some aspects. For example, wild cards in the incoming message address are not matched by default (this can be enabled by using an `OSCFunc` or `OSCdef` created with the `newMatching` method), and any OSC blobs (blocks of binary data) are always interpreted as an `Int8Array`.

4.2.3.3 OSC via TCP SuperCollider can also act as a TCP client (i.e., `NetAddr` can be used to connect to a TCP service and send and receive messages). To demonstrate this, we will need to start a service to connect to. We'll use the Unix utility *netcat*, which is named *nc* on many systems. Start *netcat* from a terminal window as a TCP server listening on a local port:

```
nc –lp 7878
```

Then use `NetAddr` to connect to the service:

```
~host = NetAddr("localhost", 7878);
~host.connect;
~host.sendMsg("/tcpTest", "tcp test message");
```

When you are done with using the service, call the disconnect method of `NetAddr`:

```
~host.disconnect;
```

OSC communication via TCP is still relatively rare in comparison to the ubiquity of the UDP protocol, but some SuperCollider subsystems use TCP.

See also Help files: NetAddr, OSCFunc, OSCdef, Server-Command-Reference, SendReply, SendTrig

4.2.4 SerialPort

In SuperCollider, communication with serial devices is done by using the `SerialPort` class, which encapsulates a connection to a specific device and supplies methods for reading, writing, and error discovery.

The `SerialPort.new` method takes a number of rather technical arguments that determine which serial port driver to use and how to configure the connection. The `port` argument is either a special device in the /dev/ directory pointing to a serial communications device, or an index to the default list of devices returned by `SerialPort.devices`. Further, `baudrate` determines the bit rate of the serial communication; possible values are all standard POSIX baud rates, and the default is 9600; `databits` configures the number of actual data bits in each transmitted symbol, with 8 as the default for most applications; `stopbit` is a flag indicating whether to transmit a single stop bit after each symbol for synchronization at the receiving device; `parity` configures the type of parity bit sent after each symbol—the default `nil` omits the parity bit, while `'even'` and `'odd'` add a parity bit such that each symbol including the parity bit contains an even or odd number of bits, respectively; `crtscts` is a Boolean flag specifying whether to use the *Request to Send* and *Clear to Send* hardware flow control lines; and `xonxoff` enables or disables software flow control. Finally, `exclusive` attempts to open the device driver in exclusive mode.

As an example, we will list the available devices, and open one of them:

```
SerialPort.listDevices;
(   // open the port
~port = SerialPort(
   "/dev/cu.usbmodem141101", // your device's name as it   appears
```

```
 in the output of
   .listDevices.
   baudrate: 9600, // must match baudrate of the device
   crtscts: true
);
);
```

Once a `SerialPort` object is created, data should be written to and read from the device from within a `Routine` (i.e., not from the top-level interpreter), so as not to block execution while performing serial communication. After using a `SerialPort` (before quitting SuperCollider), you should always close the port with `~port.close;`

The details of serial communication are highly dependent on the devices and communication protocols involved. In the next sections, we will give some simple examples of interfacing with some simple and/or common protocols.

4.2.4.1 Writing serial data The `put` method transmits a single byte of data to the device, as determined by `databits` when opening the port. The behavior of `put` is always synchronous (i.e., it blocks the currently running SuperCollider thread until the whole byte can be transmitted). The optional `timeout` parameter specifies the granularity at which write attempts are scheduled. The method `putAll` is a wrapper around `put` that writes a collection of bytes to the serial device.

Let's assume that you have a device that will read a byte from its serial port and control the brightness of an LED based on that byte.[8] To send the byte from SuperCollider, you can run

```
~port.put(255);    // maximum
~port.put(0);     // minimum

(
// fade in and out
r = Routine{
   loop{
        255.do {|i| ~port.put(i); 0.05.wait};
        255.reverseDo {|i| ~port.put(i); 0.05.wait};
   }
}.play;
);

// stop the routine again
r.stop;
```

4.2.4.2 Reading serial data For reading from the device, there are two different options: the method `next` returns either a single byte read from the serial device or `nil` (when currently there are no data to be read). The method `read` performs a blocking read, taking control of the current thread until an entire byte of data can be read.

As an example we assume you are communicating with a device that outputs data by printing them as ASCII characters, thus forming a string. The values that are measured are printed as numerals, a comma is used to divide between different values, the return character is used to indicate the end of a message, and the newline character is used to indicate the start of a message.

To read the data, we can then create a routine as shown in figure 4.4.

```
(
~routine = Routine{
var byte, str, val;
inf.do{|i|
   // start reading when we have seen a new line character:
   if(~port.read == Char.nl.asInteger, {
        str = " "; // create an empty string
        // add bytes to the string,
        // as long as we have not read the return character:
        while(
              {byte = ~port.read; byte != Char.ret.asInteger},
              {str = str ++ byte.asAscii}
        );
        // split the string based on the comma, and
        // convert each substring to an integer:
        val = str.split(Char.comma).asInteger;
        // our sound triggered and controlled
        // val[0]: push button = on/off sound
        // val[1]: pot0 = freq
        // val[2]: pot1 = amp
        if(val[0] == 1, {
        {SinOsc.ar(
              val[1].linexp(0, 1023, 400, 1600), 0,
              val[2].linlin(0, 1023, -30, -16).dbamp
        ) * Env.perc(0.01, 0.1).kr(2)}.play;
        });
   });
};
}.play;
)
```

```
// stop
~routine.stop;
~port.close;
```

Figure 4.4
Reading data from the SerialPort with a Routine.

4.2.4.3 Arduino and custom serial protocols Since 2006, Arduino has become one of the most popular physical computing devices, which gives the artist easy access to a programmable chip for interfacing analog and digital sensors such as ultrasound and infrared devices, accelerometers, custom circuits, and more. Quoting from the Arduino website:[9] "Arduino is an open-source electronics platform based on easy-to-use hardware and software. It's intended for anyone making interactive projects."

There are a number of different Arduino boards available, which connect to a host computer through a serial port (a serial device emulated over a USB link). The boards either contain an Atmel AT-mega microcontroller or an ARM CPU that can be programmed from the Arduino development environment using a superset of the C/C++ programming language. The most basic board, the Arduino Uno, features 14 digital (including pulse-width modulation and serial lines) and 6 analog (10-bit resolution each) inputs/outputs. Code can be written, compiled, and uploaded to the board's flash memory, keeping development turnaround times short and easing experimentation. Once programmed, the board can run in stand-alone mode, communicating with the outer world via its digital and analog ports, or it can communicate with other applications via its built-in serial port.

An example of how to communicate between SuperCollider and Arduino can be found in the Messenger library for Arduino.[10] The classes `Wiring` and `WiringParser` can be subclassed to define your own protocol.

Arduino's analog inputs allow for sensing real-world data in a resolution sufficient for many control applications, while the configurable digital ports can be used to connect to a wide variety of external hardware, including devices employing the digital Two Wire Interface (TWI) protocol.[11]

4.2.4.4 Error detection `SerialPort` makes error detection and recovery entirely the programmer's responsibility. The method `rxErrors` can be of help for this task: it returns the number of data bytes received from the device but not consumed by the SuperCollider application, indicating a communication error of some sort and the need for resynchronization with the sending device.

See Also
Help files: SerialPort

Quarks: Arduino, DMX (for sending control data to theatrical lights)

4.2.5 Analog Audio Sensing

In many cases, there's a very viable alternative to dedicated sensor boards and environments: use the unused analog inputs of your multichannel audio interface. The synthesis server *scsynth* provides a plethora of *Unit Generators* for massaging incoming sensor data according to the application's needs. Of particular interest are digital filters such as `Median`, `LPF`, `HPF`, and `BPF`; signal analysis UGens such as `RunningSum` and `Slope`; trigger-oriented UGens such as `Trig`, `Latch`, and `Gate`; and the whole spectrum of machine-listening plug-ins (also see chapter 15).

The UGens `SendTrig` and `SendReply` are the main means of communicating triggers and signal values to the language, where they can be appropriately acted upon. Below is a small example of how one might process data from the first audio input, extract triggers in the time domain, and send the value to the language. A possible use case is triggering actions via piezoelectrical percussion pickups:

```
(
{
   var in = RunningSum.rms(SoundIn.ar([0, 1]).sum, 10),
   thresh = MouseX.kr(0, 2), // variable threshold
   trig = in >thresh; // define trigger
   SendTrig.ar(trig, 0, in);
}.play;
);
o = OSCFunc({|. . . args| args.postln}, "/tr", s.addr);
// to remove the OSCFunc again when you are done:
o.remove;
```

4.2.6 Custom Interfaces with Bela

The Bela board[12] provides an interface to work with custom sensors and custom outputs on an embedded Linux platform (see chapter 11) with low latency: with this board, you can read out sensors with *scsynth* rather than using serial communication. For this, there are specific UGens that work only on Bela: `AnalogIn`, `AnalogOut`, `DigitalIn`, `DigitalOut`, and `DigitalIO`. After you have read in sensor data using one of these UGens, you can either use them directly to control parameters in your synthesis process or send the data back to the language using `SendReply` in your Synth definition and an `OSCFunc` in the language. The latter can be useful if you want to control higher-level interactions with your sound material (e.g., starting and stopping `Synths`).

4.3 The Modality Toolkit

The Modality Toolkit is a library (available as a Quark) to simplify the creation of individual (electronic) instruments with SuperCollider, using controllers of various kinds. The toolkit uses a unified API to interface with controllers from various sources and protocols. Currently, all of MIDI, HID, and OSC devices are supported.

The library supports many existing commercial HID and MIDI devices. Responder functions can be registered for individual elements of a controller, with such elements classified according to type and organized in a tree data structure which mimics the spatial configuration of the device. This hides from the user the particularities of the protocols involved, such as which MIDI CC number is used by a specific physical control. Additional devices can be made compatible with the library by writing a corresponding description file. This involves gathering a list with the protocol addresses (e.g., MIDI CC number) used for each physical control and deciding on a layout for the tree data structure. Modality provides the classes `HIDExplorer` and `MIDIExplorer` to help in this process: in the case of `HIDExplorer`, the description file is generated using information gathered from the low-level HID stack, while in the case of `MIDIExplorer`, the user must move all physical controls to help with element discovery.

When using the library, the first step is to discover which devices are available for use:

```
MKtl.find;
```

Executing this code will list all detected devices in the post window, for example:

```
// [["nanoKONTROL2", "SLIDER/KNOB", 10728089]]
   // Supported. Create by lookupName only if necessary:
// MKtl('midi_1_nanoko', 'midi_1_nanokontrol2');
   // Best create MKtl from desc file:
MKtl('midi_1_nanoko', "korg-nanokontrol2");
```

If the device is supported, the output will contain a line of code which can be used to create an `MKtl` for this device, in this case the Korg nanoKONTROL2.

```
k = MKtl('nk2', "korg-nanokontrol2");
```

Here, `'nk2'` is a user-provided name for future recall and `"korg-nanokontrol2"` is the name of the description file which should be used. A reference to the object is kept in the variable `k` for easy access, although another reference can always be obtained using the user-provided name by calling `MKtl('<user-provided name>')`.

If you don't currently have a Korg nanoKONTROL2 connected to the computer, it is possible to use the controller in virtual mode by creating a GUI representation, as shown in figure 4.5:

Figure 4.5
The Modality GUI representation of the Korg nanoKONTROL2 MIDI controller.

```
k.gui;
```

For debugging purposes, the output from each element can be printed to the post window using `MKtl`'s trace method or the trace button in the GUI:

```
// turn tracing on
k.trace;
// turn it off
k.trace(false);
```

Each control (i.e., knob, slider, button, etc.) on the device is represented by `MKtlElement`. Elements are organized in hierarchical order using `MKtlElementGroup`. It is possible to print on the post window a hierarchical list of all elements in order to know how to access a particular element:

```
k.postElements;
```

Below you can find an excerpt of the output for the Korg nanoKONTROL2:

```
/////// MKtl('nk2') .postElements: //////
Group: nil
0   Group: 'tr'
    0   'tr_rew'
    1   'tr_fwd'
```

```
(. . .)
1   Group: 'sl'
    0   'sl_1'
    1   'sl_2'
(. . .)
3   Group: 'bt'
    0   Group: 'S'
    0   'bt_S_1'
    1   'bt_S_2'
(. . .)
```

Single and group elements can be accessed using the `elAt` method:

```
k.elAt(\tr, \rew); // the rewind button
k.elAt(\sl, 0); // the first slider
k.elAt(\sl); // a group containing all sliders
k.elAt(\bt, \S, 0); // the first button of the top row   of buttons
```

Actions for an element (or group) are defined using the `action_` or `addAction` method of `MKtlElement`. These methods take a function as the argument, which receives a single argument, the `MKtlElement` it belongs to. One can get the current value of the element by calling the `value` method. The value returned by an `MKtlElement` is always between 0.0 and 1.0:

```
// assign an action to the third knob
k.elAt(\kn, 2).action_({|el| [el.name, el.value.round(0.0001)].
postcs});
// reset the action of this knob to nothing (nil)
k.elAt(\kn, 2).resetAction;
```

It is also possible to add an action to a group. In that case, the action will be called whenever any element of the group receives a message.

```
// add an action to the group of all knobs
(
k.elAt(\kn).action_({|el|
      "knob % value: %\n".postf(el.parent.indexOf(el), el.value)
});
);
//reset the \kn group's action to nil
k.elAt(\kn).resetAction;
```

Figure 4.6 shows the use of actions to control a very simple sound process. The synth has 4 parameters which are mapped to 4 elements: two faders and two knobs.

```
s.boot;
// using a very simple SynthDef and Synth:
(
SynthDef(\blippy, {
   var snd = Blip.ar(
        \freq.kr(440).lag(0.1),
        \numHarmonics.kr(100),
        \amp.kr(1.0).lag(0.1)) * 0.5;
   Out.ar(0,
        Pan2.ar(snd, \pan.kr(0).lag(0.1))
   )
}).add
);
// start the synth by hand first
z = Synth(\blippy, [\freq, 440, \numHarmonics, 100, \amp,   0.5, \p
an, 0]);

// create 4 control elements for it:
(
// clear all actions first
k.resetActions;
// slider 0 -> amplitude
k.elAt(\sl, 0).action_({|elem| z !? _.set(\amp, \amp.asSpec   .map
(elem.value))});

// knob 0 -> pan
k.elAt(\kn, 0).action_({|elem| z !? _.set(\pan, \pan.asSpec   .map
(elem.value))});
// slider 1 -> freq
k.elAt(\sl, 1).action_({|elem| z !? _.set(\freq, elem.value   .linl
in(0.0, 1.0, 50, 2000))});

// knob 1 -> number of harmonics
k.elAt(\kn, 1).action_({|elem| z !? _.set(\numHarmonics, elem   .va
lue.linexp(0.0, 1.0, 1, 50))});
)
```

Figure 4.6
Defining a very simple instrument with the Modality Toolkit.

It is often the case that a controller used for an instrument is not currently available. For instance, you forgot to bring it with you or it broke down. It is therefore desirable to decouple the logic of the instrument from the particular controller being used. This allows quickly reconfiguring the instrument in order to use a different controller. In the Modality Toolkit, this can be done with `MKtl`'s `addName` method, which enables referring to an existing element by a different name. In figure 4.7, `addName` is employed to enable using one of two possible controllers to play the same instrument.

```
// remove all actions
k.resetActions;
// add new names for the elements used in the instrument
(
// control synth parameters
k.addNamed(\amp, k.elAt(\sl, 0));
k.addNamed(\pan, k.elAt(\kn, 0));
k.addNamed(\freq, k.elAt(\sl, 1));
k.addNamed(\numHarmonics, k.elAt(\kn, 1));
// use play and stop buttons to create and free the synth
k.addNamed(\start, k.elAt(\tr, \play));
k.addNamed(\stop, k.elAt(\tr, \stop));
);
// give them the same actions as before.
(
k.elAt(\amp).action_({|elem| z !? _.set(\amp, \amp.asSpec   .map(el
em.value))});
k.elAt(\pan).action_({|elem| z !? _.set(\pan, \pan.asSpec   .map(el
em.value))});
k.elAt(\freq).action_({|elem| z !? _.set(\freq, elem.value   .linli
n(0.0, 1.0, 50, 2000))});
k.elAt(\numHarmonics).action_({|elem| z!? _.set(\numHarmonics,   el
em.value.linexp(0.0, 1.0, 1, 50))});

// and new functions for start and stop:
k.elAt(\start).action_({|elem|
   if(elem.value > 0) {// only start on button down
        z!? _.free;
        z = Synth(\blippy, [\freq, 440, \numHarmonics, 100,   \amp,
0.5, \pan, 0])
   }
});
k.elAt(\stop).action_({|elem|
   if(elem.value > 0) {// only stop on button down
        z!? _.free; z = nil;
```

```
    }
});
);
// To control the same instrument from a different controller evalu
ate the block of code below,
// followed by the block of code above.
(
k = MKtl(\gp, "*impact-gamepad"); // k is now the new controller
k.gui;
k.addNamed(\amp, k.elAt(\joy, \r, \y));
k.addNamed(\pan, k.elAt(\joy, \r, \x));
k.addNamed(\freq, k.elAt(\joy, \l, \x));
k.addNamed(\numHarmonics, k.elAt(\joy, \l, \y));
k.addNamed(\start, k.elAt(\bt, \5));
k.addNamed(\stop, k.elAt(\bt, \7));
);
```

Figure 4.7
Using named elements to decouple the instrument from the controller.

It is also possible to add multiple actions to an element using `addAction`:

```
(
k = MKtl('nk2');
k.resetActions;
f = {|elem| ("1: "++elem.value).postln};
g = {|elem| ("2: "++elem.value).postln};
k.elAt(\sl, 0).addAction(f);
k.elAt(\sl, 0).addAction(g);
);
```

Whenever the slider is moved, both actions will run. To remove an action, a reference to the corresponding function is needed:

```
k.elAt(\sl, 0).removeAction(f);
```

Now, only the second action will run.

Some controllers are capable of receiving data which can be used, for instance, to reposition faders or encoders or change the state of LEDs. As an example, the controller Behringer BCR2000 has several encoders and buttons whose internal value can be changed via MIDI messages. After initializing the `MKtl`, you can determine which elements accept an output value:

```
k = MKtl('bcr', "behringer-bcr2000");
k.gui;
// list elements which can send a value back to the controller
k.outputElements;
```

The following code repositions the first knob of the top row to its halfway position:

```
k.elAt(\kn,0,0).value_(0.5);
```

For additional information, see Baalman et al. (2014) and the library documentation.

4.4 Using the Data

As in electronic music, the input controls are disconnected from the acoustic output, the connection between them necessarily becomes a part of the compositional process. There are several approaches that can be taken toward this problem, and there is no definite best choice for every case.

An important consideration in creating this mapping is the representation of music that you will be interacting with—what sort of processes will you be controlling, and at which time scale will you be controlling them? Will you control the fine nuances of texture of the sound, will you trigger single notes, are you manipulating the playback of phrases or notes, or are you scrubbing sound samples?[13]

Another major distinction is whether you are using an instrumental approach (e.g., you are trying to create a musical instrument, similar to an acoustic instrument) or a compositional approach (e.g., you are using interactive technology to create an interaction proposition for the performer).[14]

The exact approach you choose depends on what you try to create: if you are creating an interactive sound installation, you have to create a connection between input and sonic result that is easy to understand for a novice user, as well as engaging, so that it maintains interest even after the initial exploration. If you are creating a performance instrument, you may want to think about what certain gestures mean; gestures themselves may be made up of data arising from several inputs at once. If you are using sensor technology in dance, you may not want to force movement in a certain way to create a certain sound, but rather to let the sound evolve based upon the movements that occur.

You may need to scale your data or map it to a different range. The `ControlSpec` class can be useful for this. Your data may need to be massaged or filtered before they will be useful. We've noted some of the UGens that can be used to do this on the server side in the section above on analog audio input. In the language, you might try storing incoming data in `Arrays` and using methods such as `median` to limit the effect of stray

values, or `maxItem` to get the highest value. The `SenseWorld` Quark also provides some useful techniques for massaging and filtering incoming data.

We recommend taking several development cycles, getting feedback from audience members or test users, and practicing with the interface you have created (including the mapping). Based on the results of this, you can more reliably determine what you might want to change. But at some point in a cycle, you have to stop changing the instrument and start learning to play it in order to achieve full control over it (training your muscle memory).

Implicit methods for finding connections between input controls and sound parameters can be found in the Quark `Influx`, which provides methods to randomize the connections between controls and parameters, and then provides methods to explore, store, and restore settings of these connections.

See also the Help files `ControlSpec`, `Array`, `ArrayedCollection`, `SequenceableCollection`. Quarks: SenseWorld, Influx, adclib (especially the `MFunc` class), mmPresetInterpolator

4.5 Conclusions

In this chapter, we have reviewed several ways to get data in and out of SuperCollider using the protocols OSC, serial, HID, and MIDI. We have suggested some directions in which to look for your own way to deal with the data. With all this in mind, the extensive programming capabilities within SuperCollider should enable you to realize your ideas.

Notes

1. Processing: https://processing.org.
2. Pure Data is a real-time graphical programming environment for audio, video, and graphical processing (https://puredata.info).
3. MIDI: https://www.midi.org.
4. Due to the underlying ALSA MIDI framework on Linux, it is possible for SuperCollider to open several MIDI input and output ports. The `MIDIClient.externalSources` and `MIDIClient.externalDestinations` methods are convenient methods to filter out SuperCollider's own ports.
5. You can also use methods of the low-level class MIDIIn, but these are harder to use and provide no methods for filtering the MIDI messages.
6. OpenSoundControl portal: http://opensoundcontrol.org.
7. If for some reason *sclang* cannot open port 57120, it will try to open port 57121, and one higher again if it cannot open that port, until it finds one that is available. You can always check which port *sclang* is currently using with the code `NetAddr.langPort`.
8. Example code for Arduino corresponding to the examples in these sections is available in the online repository that comes with this book. The website https://electronics.koncon.nl/supercollider/ was used as a basis for the simple instances and provides examples that go a step further as well. Also, the site includes diagrams of how to connect the LED and the sensors to the Arduino.
9. Arduino community website: https://www.arduino.cc.
10. See the website https://playground.arduino.cc/Code/Messenger/.

11. This was originally introduced and patented by the Philips Corporation under the name "I²C bus" in 1981.
12. Bela's website can be found at https://bela.io.
13. See, for example, Malloch et al. (2006).
14. The book *Composing Interactions—An Artist's Guide to Building Expressive Interactive Systems* (Baalman 2022) covers all the steps from concept to touring with an interactive work and is recommended for further reading on this topic.

References

Baalman, M. A. J. 2022. *Composing Interactions—An Artist's Guide to Building Expressive Interactive Systems*. Rotterdam, Netherlands: V2_ Publishing.

Baalman, M. A. J., T. Bovermann, A. de Campo, and M. Negrão. 2014. "Modality." In *International Computer Music Conference Proceedings*, 2014, pp. 1069–1076. Athens. Available from http://hdl.handle.net/2027/spo.bbp2372.2014.165.

Malloch, J., D. Birnbaum, E. Sinyor, and M. M. Wanderley. 2006. "Towards a New Conceptual Framework for Digital Musical Instruments." In *International Conference on Digital Audio Effects (DAFx-06)*, Montreal.

Puckette, M. 2007. *The Theory and Technique of Electronic Music.* Hackensack, NJ: World Scientific Publishing.

Reas, C., and B. Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists.* Cambridge, MA: MIT Press.

Wright, M., A. Freed, and A. Momeni. 2003. "OpenSoundControl: State of the Art 2003." In *Proceedings of the 2003 International Conference on New Interfaces for Musical Expression* (NIME-03), pp. 153–159. McGill University, Montreal.

II ADVANCED TUTORIALS

5 Programming in SuperCollider

Iannis Zannos

5.1 Introduction

This chapter provides an introduction to the SuperCollider programming language from a more technical viewpoint than David Cottle's beginner's tutorial (chapter 1). Some material presented there and elsewhere in this book is also covered here, but in a more methodical and exhaustive manner. Although I try to convey programming skills without presupposing any previous knowledge of programming languages and compiler technology, some of the more advanced programming concepts in this chapter can take a little getting used to. The explanations provided here aim to be complete within the space allowed, but readers new to object-oriented programming may wish to seek out a good general introductory text to accompany their explorations (http://en.wikipedia.org/wiki/Object-oriented_programming isn't bad!). Musicians new to computer music may prefer to start earlier in the book and return here only once they have acquired some basic familiarity with the language; for experienced computer musicians new to SuperCollider, or experienced programmers new to audio, this chapter may be the preferred entry route. For everyone, it should provide a useful reference on the SuperCollider language.

Mechanisms underlying the interpretation and execution of programs and the programming concepts of SuperCollider will be explained. This will serve as a basis for understanding how to write and debug effectively in SuperCollider. We will consider the following questions:

- What are the basic concepts underlying the writing and execution of programs?
- What are the fundamental program elements in SuperCollider?
- What are objects, messages, methods, and classes, and how do they work?
- How are classes of objects defined?
- What are the characteristic techniques of object-oriented programming, and how are they applied in SuperCollider?

SuperCollider employs syntactic elements from C, C++, Java, Smalltalk, and Matlab, creating a style that is both concise and easy to understand for programmers who know

one of these common programming languages. A summary of the SuperCollider language syntax is given in the appendix of this book.

5.2 Fundamental Elements of Programs

5.2.1 Objects and Classes

The language of SuperCollider implements a powerful method for organizing code, data, and programs known as *object-oriented programming (OOP)*. SuperCollider is a pure OOP language, which means that all entities inside a program are some kind of object. It also means that the way these entities are defined is uniform, as are the means for communicating with them.

5.2.1.1 Objects *Objects* are the basic entities that are manipulated within programs. They bundle together data and methods that can act on that data (a musical scale object might store the pitches in the scale and a method to play back those pitches up and down as a sequence of notes in order). In the simplest case they might look like a single number or letter (a character), but they still respond to a number of methods for acting on themselves (return the negative of the current number, return the ASCII key code for the letter) or to more complex methods combining multiple objects (add this number to another to make a third number, combine this letter with another to make a 2-letter word). In practice, there are two main types of objects, categorized according to how their internal contents are organized: objects with a fixed number of internal slots for storing data and objects with a variable number of slots. The generic term for the latter type of object is *collection*. Collections prove useful for handling big sets of data, such as a library of musical tunings or a mass of partial frequency, amplitude, and phase data for additive synthesis.

Some examples of objects include those shown in figure 5.1. You can see various different objects in the code listing, from simple numbers and letters to more abstract types that will be explained in more detail during the course of this chapter.

```
1       // the Integer number 1
1.234   // the floating-point (Float) number 1.234
$a      // the character (Char) a
"hello" // a String (an array of characters)
\alpha  // a Symbol (a unique identifier)
'alpha 1'   // another notation for a Symbol
100@150 // a Point defined by coordinates x, y
[1, \A, $b]    // an Array containing 3 elements
(a: 1, b: 0.2) // an Event
{10.rand}       // a Function
```

```
String  // the Class String
Meta_String     // the Class of Class String
```

Figure 5.1
Some objects.

5.2.1.2 Classes A *class* describes the attributes and behavior that are common to a group of objects. All objects belonging to a class are called *instances* of that class. For example, all integer numbers (e.g., 0, −1, and 50) are instances of the class `Integer`. All integers are able to perform arithmetic operations on other numbers; therefore, the class Integer describes—among other things—how integers perform arithmetic operations. Instances are created as literals (objects with a "literal" representation in code, such as `1`, `-10`, `$a`, `\a`; more on this later in this chapter) with one of the shortcut constructor syntax forms (e.g., `{}`, `()`, `a@b`, `a->b`) or by sending a special message to a class that demands an instance. The most common message for creating instances is `new`, which can be omitted for brevity: `Rect.new(10, 20, 30, 40)` is equivalent to `Rect(10, 20, 30, 40)`, and both create a rectangle with the specified numbers determining its size and position.

A class can inherit properties and behavior from another class, called its *superclass*. A class that inherits properties is a *subclass* of the class from which it inherits. The mechanism of inheritance is central in object-oriented programming for defining a hierarchical family tree of categories that relate to each other. This promotes sharing of common functionality while allowing the specialization of classes for particular tasks. (We'll return to this later.)

5.2.2 Literals

Literals are objects whose value is represented directly in the code (rather than computed as a result of sending a message to an object). Literals in SuperCollider are the following:

- Integers (e.g., `−10`, `0`, `123`) and floating-point numbers (e.g., `−0.1`, `0.0`, `123.4567`), which can be in exponential notation (e.g., `1e4`, `1.2e−4`); alternative radices up to base 36 (e.g., binary for 13 is `2r1101`, and the hexadecimal for 13 is `16rD`); or combined with the constant pi (e.g., `2pi`, `−0.13pi`).
- Strings, enclosed in double quotes: `"a string"`
- Symbols, enclosed in single quotes (`'a symbol'`) or preceded by \: `\a_Symbol`.
- Literal Arrays: immutable Arrays of literals declared by prepending the number sign `#` (e.g. `#[1, 2.3, 5]`).
- Classes: A class is represented by its name. Class names start with a capital letter. Characters (instances of `Char`), a single character preceded by the dollar sign $ (e.g.,

`$A`, `$a`); nonprinting characters (e.g., newline and tab) and backslash are preceded by a backslash (e.g., `$\n`, `$\t`, `$\\`).

• Variables and constants (see also the SuperCollider Help file on Literals and the appendix).

5.2.3 Messages and methods

To interact with an object, one sends it a *message*. For example, to calculate the square of a number, one sends the message squared:

```
15.squared // calculate and return the square of 15
```

The object to which a message is sent is called the *receiver*. In response to the message, the receiving object finds and runs the code that is stored in the *method* which has the same name as the message, then returns a result to the calling program, which is called the *return value*. In other words, a method is a function stored under a message name for an object that can be recalled by sending that object the message's name (see figure 5.2).

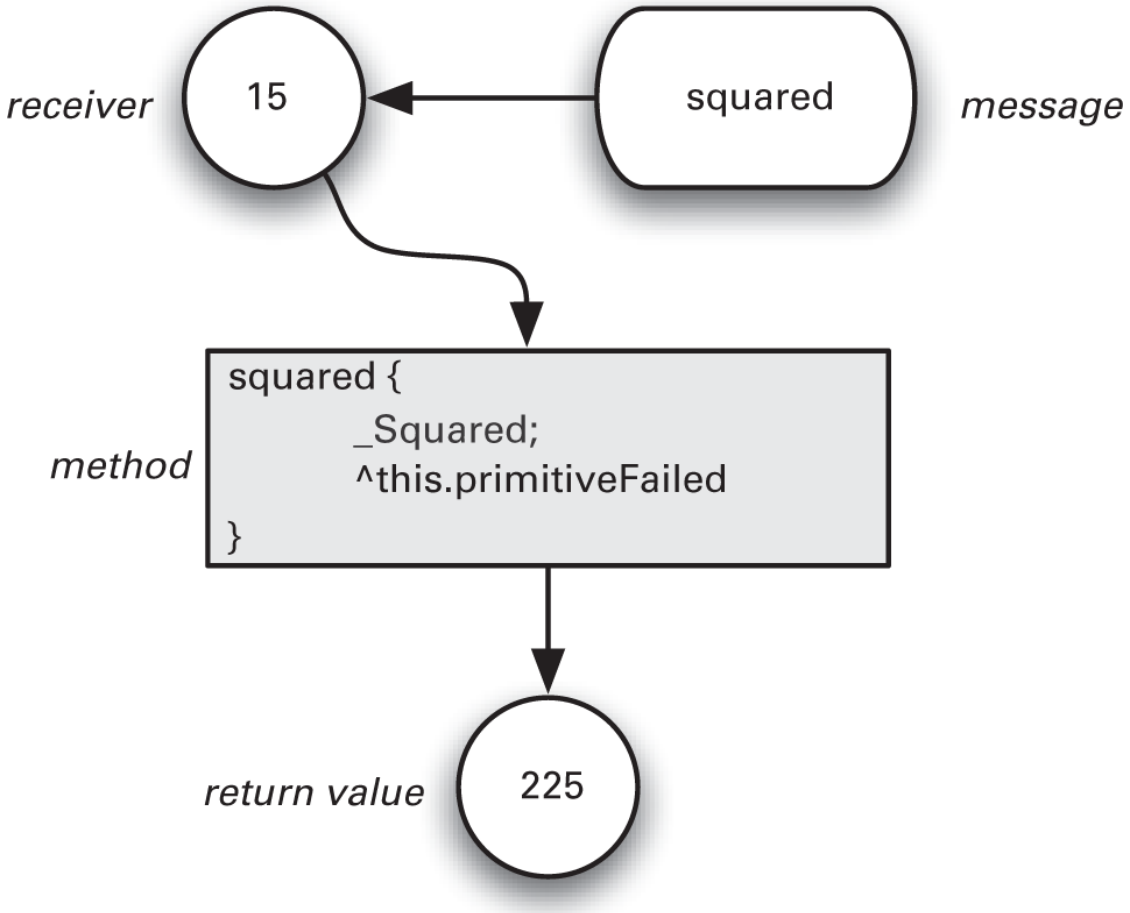


Figure 5.2
Receiver, message, method, return value.

Instance methods operate on an instance (such as the integer `1`), and *class methods* operate on a class. As noted above, the most commonly used class method is `new`, which is used to create new instances.

An alternative way of writing a message is in C-style or Java-style function-call form. The above example can also be written as follows:

```
squared(15) // calculate and return the square of 15
```

SuperCollider often permits one to choose among different writing forms for expressing the same thing. Two main criteria that programmers take into account in choosing are readability and conciseness.

5.2.3.1 Chaining messages It is possible to write several messages in a row, separated by dots (.), like the one below:

```
Server.local.boot    // boot the local server
```

or this:

```
Server.local.quit    // quit the local server
```

When chaining messages, each message is sent to the object returned by the previous message (the previous *return value*). In the examples above, `Server` is the class from which all servers are made. Among other things, it holds by default two commonly used servers, the *local server* and the *internal server*, which can be obtained by sending it the messages `local` and `internal`, respectively. The objects and actions involved are the following:

```
Server // the class Server
// message local sent to Server returns the local server:   localho
st
Server.local
Server.local.boot    // the message boot is sent to the local   ser
ver
```

5.2.3.2 Performing messages In some cases, the message to be sent to an object may change, depending on other conditions. When the message is not known in advance, the messages `perform` and `performList` are used, which allow an object to perform a message passed as an argument:

```
Server.local.perform(\boot)   // boot the local server
// boot or quit the local server with 50% probability of either:
Server.local.perform([\boot, \quit].choose)
```

Here, `performList` permits one to pass additional arguments to the message in `Array` form. Thus `Rect.performList(\new, [0, 10, 200, 20])` is equivalent to `Rect.new(0, 10, 200, 20)`. (See also figure 5.22.)

5.2.4 Arguments

The operation of a message often requires the interaction of several objects. For example, raising a number to some power involves two numbers: the base and the exponent. Such additional objects required by an operation are sent to the receiver as *arguments* accompanying the message. Arguments are enclosed in parentheses after the message:

```
5.pow(8)   // calculate the 8th power of 5
```

If several arguments are involved, they are separated by commas:

```
// construct an Array of 5 elements starting at 10 and incrementing
by 10


Array.series(5, 10, 10)
```

The same is true in function-call format:

```
series(Array, 5, 10, 10)
```

If the arguments are provided as one collection containing several objects, they can be separated into individual values passed in sequence by prepending the `*` sign to the collection:

```
Array.rand(*[5, -10, 10])
```

is equivalent to

```
Array.rand(5, -10, 10)
```

This can be useful when one wants to provide arguments as a collection that was created in some other part of the program. The next example shows how to construct an Array of random size between three and ten with elements whose values have a random range, with three as the lowest and ten as the highest possible value:

```
Array.rand(*Array.rand(3, 3, 10))
```

When the only argument to a message is a function, the parentheses can be omitted:

```
10.do {10.rand.postln} //function as sole argument to a message
```

5.2.4.1 Argument forms for the implied messages at and put When square brackets are appended to an object, they imply the message `at` or `put` (this follows the C or Java syntax for Array indexing). Thus `[1, 5, 12][1]` is equivalent to `[1, 5, 12].at(1)`, and `()[\a] = pi` is equivalent to `().put(\a, pi)`.

5.2.4.2 Argument keywords When calling a function, argument values must be provided in the order in which the arguments were defined (see section 5.4.4.1). However, when only a few of many arguments of a function need to be provided, one

can specify those arguments by name in "keyword" form; for instance, if the name of the argument provided is `freq`, the call is `foo.value(freq: 400)`. Similarly, the `kr` method for `XLine` takes the arguments `start`, `end`, `dur`, `mul`, `add`, and `doneAction`. To accept the defaults for `mul` and `add`, one can write `XLine.kr(100, 100, 10, doneAction: 2)`. As a result, `start`, `end`, and `dur` get the values 100, 1000, and 10, respectively; `doneAction` gets the value 2; and `mul` and `add` rely on their default values 1 and 0, respectively. (See figure 5.3.)

```
// Boot the default server first:
s.boot;
// Then select all lines between the outermost parentheses and run:
(
{
   Resonz.ar(GrayNoise.ar,
        XLine.kr(100, 1000, 10, doneAction: 2),
        XLine.kr(0.5, 0.01, [4, 7], doneAction: 0)
   )
}.play
)
// further examples:
{WhiteNoise.ar(EnvGen.kr(Env.perc, timeScale: 3, doneAction: 2))}
.play;
{WhiteNoise.ar(EnvGen.kr(Env.perc, timeScale: 0.3, doneAction:
 2))}.play;
```

Figure 5.3
Keyword arguments.

5.2.5 Binary Operators

SuperCollider uses signs from mathematics, logic, and other programming languages, such as + (addition), − (subtraction), and `&` (binary "and"). These are called *binary operators* because they operate on two objects. For example, ++ joins two SequenceableCollections: `[\a, \b] ++ [1, 2, 3]`.

Furthermore, any message that requires just one argument can be written as a binary operator by adding `:` to the name of the message. Thus, `5.pow(8)` can also be written as `5 pow: 8`. With this and other syntax shortcuts included in this chapter, there may seem to be a bewildering variety of alternatives available. SuperCollider supports a few different common programming syntaxes, but the "dot" notation is most common, and with practice, you can pick up additional syntax as you code and gain experience. Further details are available in the Syntax Shortcuts Help file.

5.2.6 Precedence Rules and Grouping

When one combines several operations into one expression, the final result may depend on the order in which those operations are executed. Compare, for example, the expression `1 + (2 * 3)`, whose value is 7, with the expression `(1 + 2) * 3`, whose value is 9. The order in which operations are executed is determined by the precedence of the operators. The precedence rules in SuperCollider are simple but differ somewhat from those used in mathematics:

- Binary operators are evaluated in strict left-to-right order. Thus, the expression `1 + 2 * 3` is equivalent to `(1 + 2) * 3`, not to `1 + (2 * 3)`.
- Message passing, as in `receiver.message(arguments)` or in `collection[index]`, takes precedence over binary operators. Thus, in `10 * (1..3).addAll([0.1, 0.2, 0.3])`, the elements of `[0.1, 0.2, 0.3]` are first appended to `[1, 2, 3]`, and then the resulting new Array is multiplied by 10.
- To override the order of precedence, one uses parentheses `()`. For example:

```
1 + 2 * 3 // Left-to-right order of operator evaluation. Result: 9.
1 + (2 * 3) // Forced the evaluation of * before that of +. Result:
7.
```

Figure 5.4 illustrates the effects of grouping by parentheses and message passing.

```
(1 + 2).asString).interpret    // = 3
"1" ++ "2". interpret      // 12: 2 is translated to string by ++
("1" ++ "2").interpret     // 12
(1.asString ++ 2.asString).interpret  // 12
"1 + 2". interpret  // 3
(1.asString ++ "+2"). interpret        // 3
(1 + 2).interpret // error: interpret not understood by   Integer 3
```

Figure 5.4
Grouping and precedence.

5.2.7 Statements

The single-line code examples introduced above normally constitute parts of larger programs that include many lines of code. The smallest stand-alone elements of code are called *statements*.[1] One creates programs by grouping sequences of statements. When a program contains more than one statement, the individual statements are separated by a semicolon. The last statement at the end of a program does not need to have a semicolon, since there are no more statements to separate it from. Figure 5.5 contains three statements.

```
(
a = 5;
5 do: {a = a + 10; a.postln};
Post << "The value of variable 'a' is now" << a << "\n";
)
```

Figure 5.5
Statements.

The first statement (`a = 5;`) assigns the value 5 to variable `a`. The second statement (`5 do: {a = a + 10; a.postln};`) repeats a function five times that assigns to `a` its previous value incremented by 10 and posts the new value of `a` each time. The third statement (`Post << "The value of variable 'a' is now "<< a << "\n";`) posts the new value of `a`.

It is important to distinguish between the lines of code text in a document window as seen by a human programmer, and the part of the code that SuperCollider processes as a program when the programmer runs a selected portion of that code. SuperCollider does not run the whole code in the window, but only the code that was selected; or, if no code is selected, the line on which the cursor is currently located. Every time that one runs a piece of code, SuperCollider creates and runs a new program that contains only the selected code. Code that is meant to be run as a whole is usually indicated by enclosing it in parentheses. This is useful because one can select it easily, typically by double clicking to the right of an opening parenthesis.

5.3 Variables

A *variable* is used to store an object that will be used in other parts of a program. One way to visualize variables is as containers with labels. The name of the variable is the label pointing to the container. (See figure 5.6.)

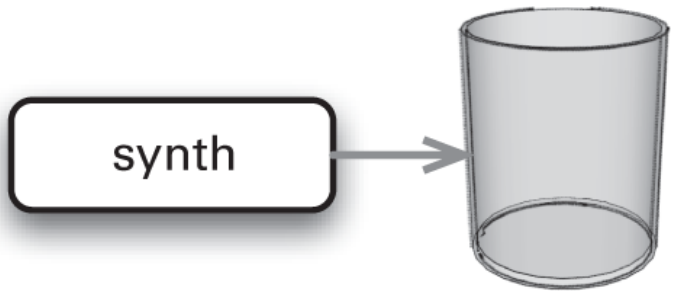


Figure 5.6
Variable as a label pointing to a container.

One creates variables by declaring them with the prefix `var`. Several variables can be declared in one var statement. Variables may be declared only at the beginning of a function (or a selected block of code, which is essentially the same thing; see section 5.4.3).

```
var window; // create a variable named 'window'
// rest of program follows here
```

When a variable is first created, it is empty, so its value is represented by the object `nil`, which is the object for no value (see figure 5.7):

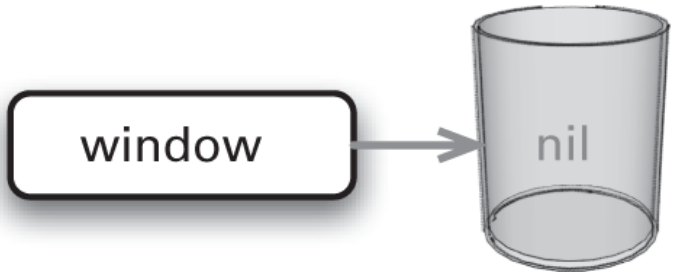


Figure 5.7
Here, `nil` stands for the contents of an empty variable.

```
(
var window; // create a variable named 'window'
window.postln;  // post the contents of variable 'window' (nil)
)
```

One cannot run the lines of a program that use a declared variable separately; one must always run the code as a whole. This is because the variables declared in the beginning of a function disappear from memory as soon as the function that declared them finishes unless other functions within that function also use them. In figure 5.7, running the line `window.postln;` alone produces the error message `Variable 'window' not defined.`

To store an object in a variable, use the assignment sign =. For example, after storing a Window in the variable window, one can send it messages to change its state, as well as use it as an argument to other objects. (See figure 5.8.)

```
(
// A window with a button that posts: "hello there!"
var window, button;
// create a GUI window and store it in variable window
```

```
window = Window.new("OLA!", Rect(200, 200, 120, 120));
// create a button in the window and store it in variable button
button = Button.new(window, Rect(10, 10, 100, 100));
button.states = [[" 'ALLO"]];    // set one single label for the
 button
button.action = {"hello there!".postln}; // set the action of   the
button
window.front;      // show the window
)
(
var bounds = Rect(10, 20, 30, 50), x = 100, y = 200;
bounds.width.postln;// post the width of a rectangle
bounds.moveTo(x, y);  // move the rectangle to a new position
)
```

Figure 5.8
Variables can store objects that need to be used many times.

In this example, the variable window was indispensable to specify in which window the button was going to appear.

5.3.1 Variable Initialization

The assignment sign (=) can be used in a declaration statement to initialize the value of a variable.

5.3.2 Using Variables

The object stored in a variable remains there until a new assignment statement replaces it with something else. Variables also are often used as temporary placeholders to operate on a changing choice from a set of objects. Figure 5.9 is an example that makes extensive use of variables to create a chain of upward- and downward-moving runs of short tones.

```
(
// execute this first to boot the server and load the synth definit
ion
s.waitForBoot({
   SynthDef("ping", {| freq = 440 |
        Out.ar(0,
              SinOsc.ar([freq, freq * (4/3)], 0,
                  EnvGen.kr(Env.perc(0.05, 0.3, 0.1, −4),   doneAct
ion: 2)
              )
```

```
        )
    }).add;
});
)

(
// execute this next to create the sounds
var countdown = 100;
var note = 50;
var increment_func, decrement_func;
var action;
increment_func = {
    note = note + [2, 5, 7, 12].choose;
    if (note > 100) {action = decrement_func};
};
decrement_func = {
    note = note-[1, 2, 5, 7, 12].choose;
    if (note < 50) {action = increment_func};
};
action = increment_func;
{
    countdown do: {
        Synth("ping", [\freq, note.midicps]);
        action.value;
        0.1.wait;
    }
}.fork;
)
```

Figure 5.9
Variables can point to different objects during a process.

5.3.3 Instance Variables

An *instance variable* is a variable that is contained in a single object. Such a variable is accessible only inside instance methods of that object unless special code is written to make it accessible to other objects. For example, objects of class `Point` have two instance variables, *x* and *y*, corresponding to the coordinates of a point in 2-dimensional space. (see figure 5.10):

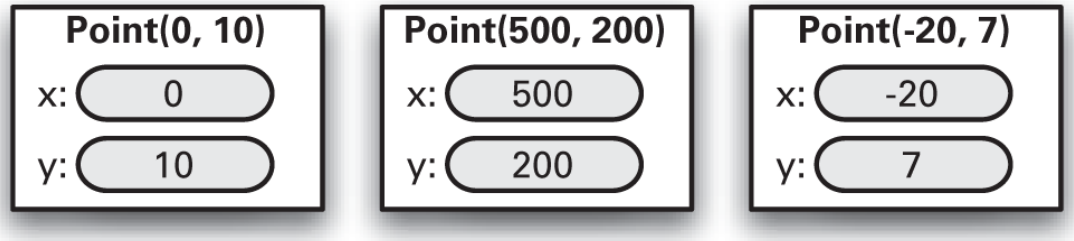


Figure 5.10
Three instances of `Point` with their instance variables.

```
(
var point = Point(0, pi);
point.x.postln; point.y.postln; point.y == pi;
)
```

5.3.4 Class Variables

A *class variable* is defined once for the class it belongs to and is shared with all its subclasses. It is accessible to class methods and to instance methods of its class and all its subclasses. For example, the class variable `allEnabled` of `OSCFunc` holds all currently active instances of `OSCFunc`. Its instance method `free` disables and removes it. In that way, the system keeps track of all OSCFuncs that are active, and checks every OSC message received to see if it matches any of the OSCFuncs contained in all. One can write `OSCFunc.allEnabled do: _.free` to remove all currently active OSCFuncs.

5.3.5 Environment Variables

Environment variables are preceded by a tilde (`~`) . For example, `~a = pi`. These reference the value of a named variable in the current `Environment`, a special holding place for data. They do not need to be declared, but they are instantly added to the Environment when assigned. An Environment is a kind of `Dictionary` that represents a set of bindings of values to names; that is, the Environment variables. These bindings differ from those created by normal variable declarations, in that they have a less limited scope (though not truly “global” variables in the traditional sense, they can sometimes be treated as such), and they can be modified more easily (see section 5.6.7 for more details).

The relationship between Environment variables and the `Environment` that contains them can be seen by printing the current Environment. (See figure 5.11.)

```
// run each line separately:
currentEnvironment;    // empty if no environment variables have
```

```
 been set
~alpha = pi;    // set env. variable ~alpha to pi
currentEnvironment;    // see current Environment again: ~alpha   i
s set
~freq = 800;    // set another environment variable
Server.local.boot;
{LFNoise0.ar(~freq, 0.1)}.play; // use an environment variable
// setting an environment variable to nil is equivalent to   removi
ng it:
~alpha = nil;
currentEnvironment;    // alpha is no longer set
```

Figure 5.11
currentEnvironment.

5.3.6 Variables with Special Uses

The variables described in this section provide access to objects that are useful or indispensable, but they either cannot be accessed by conventional programming within the SuperCollider class system or need to be accessed by all objects in the system.

5.3.6.1 Interpreter variables The class `Interpreter` defines 26 instance variables whose names correspond to the lowercase letters `a` to `z`. Since all code evaluated at runtime is run by an instance of `Interpreter`, these variables are accessible within that code without having to be declared. However, this works only when evaluating code from outside of class definitions, that is, with code selected to be run by the `Interpreter`). The following example can be executed one line at a time:

```
s.boot
n = {| freq = 400| LFDNoise1.ar(freq, 0.1)}.play; // store   a synt
h in n
n.set(\freq, 1000);  // set the freq parameter of the synth   to 10
00
n.free;  // free the synth (stop its sound)
```

5.3.6.2 Pseudovariables Pseudovariables are not declared anywhere in the SuperCollider library but are provided by the compiler. They are the following:

• `this` represents the object that is running the current method. In runtime code, this is always the current instance of `Interpreter` (see also section 5.4.3.1). Thus, one can run `this.dump` to view the contents of the current Interpreter instance, including the variables *a* – *z*.

• `thisProcess` is the process that is running the current code. It is always an instance of `Main`. Although rarely used, some possible applications are to send the current instance of Main messages that affect the entire system, such as `thisProcess.stop` (equivalent to pressing Cmd/Ctrl-.), or to access the Interpreter variables from any part of the system (`thisProcess.interpreter.a` accesses the Interpreter variable *a*).

• `thisMethod` is the method within which the current statement is running. One can use this in debug messages to print the name of the method where some code is being checked. For example, `[this, this.class, thisMethod.name].postln`. `thisFunction` is the innermost function within which the current statement is running. It is indispensable for recursion in functions (see section 5.4.9). Here, `thisFunctionDef` is the definition of the innermost function within which the current statement is running. The function definition contains information about the names and default values of arguments and variables. Section 5.4.10 briefly discusses its uses.

• `thisThread` is the thread running the current code. A `Thread` is a sequence of execution that can run in parallel with other threads and can control the timing of the execution of individual statements in the program. Examples of the use of thisThread are found in the `Pstep` and `Pseg` classes, where it is employed to control the timing of the thread.

One special case: the keyword `super` redirects the message sent to it to look for a method defined in the superclass of the object in which the method of the current code is running. This is not a variable at all because one cannot access its values but can only send it a message. Here, `super` is used to extend a method in a subclass. For example, the class method `new` of `Pseq` extends the method `new` of its superclass `ListPattern`, which in turn extends the method `new` of `Object`. This means Pseq's new calls `super.new`—thereby calling method new of `ListPattern`—but adds some statements of its own. In turn, `ListPattern` also calls `super.new`—thereby calling the method new of `Object`—but again adds some stuff of its own.

5.3.6.3 Class variables of Object The following variables are class variables of class `Object`. Since all objects are instances of some subclass of Object, they have access to these variables, and thus these variables are automatically accessible everywhere.

• `currentEnvironment` is the `Environment` being used right now by the running program. This can be changed by the programmer.

• `topEnvironment` is the original `currentEnvironment` of the Interpreter instance that runs programs in the system. It can be accessed independently of `currentEnvironment`, which changes in response to Environment's `use` or `make` methods. (See figure 5.12.)

```
(
~a = "TOP";     // store "TOP" in ~a, top environment
(a: "INNER") use: {// run function in environment with   ~a = "INNE
R"
   currentEnvironment.postln; // show the current environment
   topEnvironment.postln;      // show the top environment   (diffe
rent!)
   ~a.postln // show ~a's value in current environment
};
~a;     // show ~a's value in top environment
)
```

Figure 5.12
`topEnvironment` versus `currentEnvironment`.

In this context, `uniqueMethods` holds a dictionary that stores unique methods of objects. *Unique methods* are methods that are defined not in a class, but only in a single instance. This can be used for extending instances and/or rapid prototyping. (See chapter 8.) For example:

```
(
// create 2 windows and store them in variables p, q
#p, q = [100, 400].collect {|i|
  Window(i.asString, Rect(i, i, 200, 200)).front
}
)
// add a unique method to p only
p.addUniqueMethod(\greet, {|w| w.name = "Hello!"});
p.greet;  // p understands 'greet'
q.greet;  // but q does not understand 'greet'
```

In this, `dependantsDictionary` holds a dictionary that stores *dependants* of objects. A dependant of an Object *o* is any object that needs to be notified when *o* changes in some way. To notify the dependants of an object that the object has changed, one sends the message `changed`. Details of this technique are explained in section 5.7.7.

5.3.7 Variables Versus References

A *variable* is a container with which one can do only two things: store an object and retrieve that object. One cannot store the container itself in another container, so it is not possible to store variable *x itself* in another variable *y*. As the following example shows, what is stored is the *content* of variable *x*. When the content of variable *x* is changed, the previous content still remains in variable *y*. (See figures 5.13 and 5.14.)

```
var aref, cvar;
aref = Ref.new; // first create the reference and store it   in are
f
cvar = aref; // then store the contents of aref in cvar
aref.value = 10;  // change the value of the reference in aref
cvar.value.postln;    // and retrieve that value from cvar

(
var alpha, beta, gamma;
gamma = alpha;  // storing variable alpha in gamma only stores nil
alpha = 10;     // store 10 in alpha. . .
gamma.postln;   // but the value of gamma remains unchanged
alpha = beta;   // so one cannot use gamma as 'joker'
beta = 20;       // to switch between variables alpha and beta.
gamma.postln;   // gamma is still nil.
)
```

Figure 5.13
Variables store only values, not other variables.

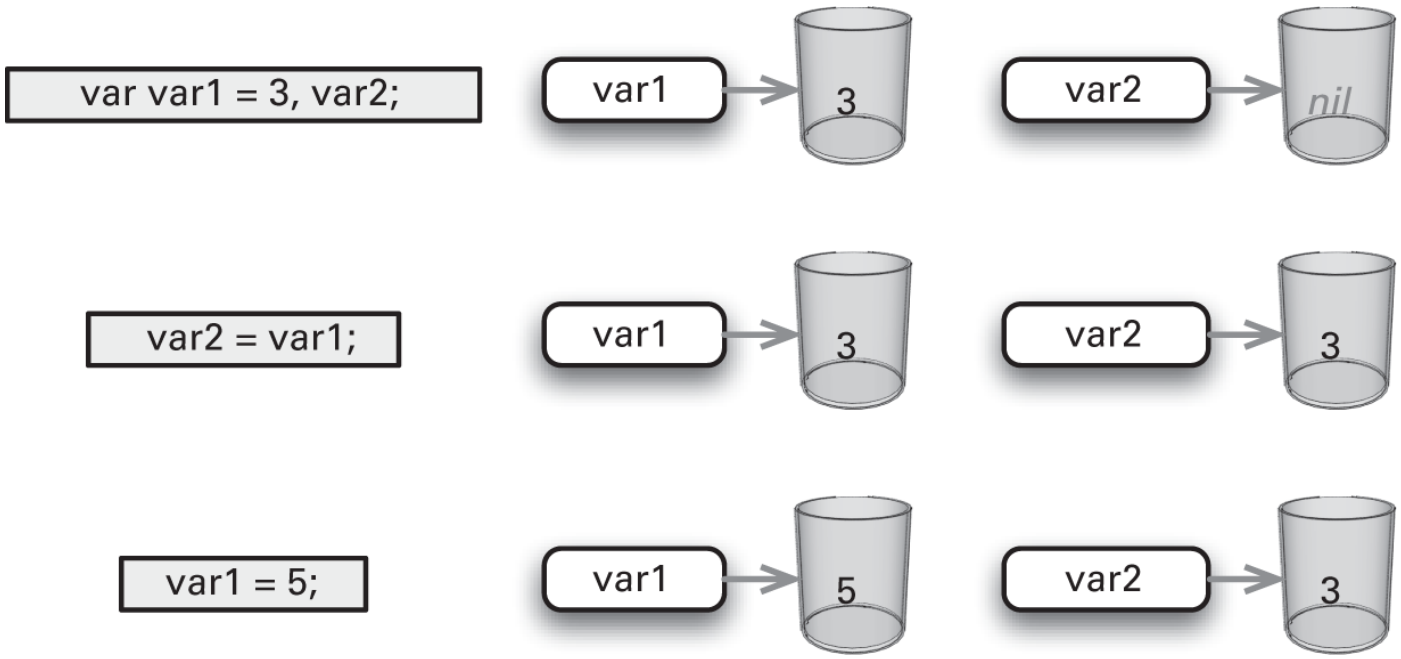


Figure 5.14
Assigning a variable to another variable stores its contents only.

To store a container in a variable, one uses a reference object:

5.4 Functions

A `Function` is an object representing a bit of code that can be evaluated from other code at runtime. In this sense, it is like a miniature program. One can "package" code that does something useful inside a function and then run that function wherever one wants to do that thing, instead of writing the same code in different places. The code that creates a function is called the *definition* of the function. When a program runs a function, it is said to *call* or *evaluate* that function.

To package code into a function, one encloses it in braces `{}`:

```
{1 + 1} // a function that adds 1 to 1
```

This creates a function object or, in other words, it *defines* a function. To run the function, one sends it the message value:

```
{1 + 1}.value // evaluate {1 + 1}
```

This is called "function evaluation."

5.4.1 Return Value versus Side Effect

The use of the term "evaluate" here comes from the idea of requesting a value that is computed and returned by the function for further use. The *return value* of a function is the value of the last statement that is computed in the function. However, in many cases, one calls a function not to obtain a final value, but rather to start a process that will result in some change, such as to create sounds or to show graphics on the screen. For example, `{10.rand}` provides a random number between 0 and 9 as a return value, and `{Window.allWindows do: _.close}` closes all GUI windows. It is the effect of the latter, rather than the return value, that matters.

This is also true for methods. In the example presented in section 5.2.3.1, `Server.local.boot`, the message `local` is sent to the class `Server` to obtain the object representing the local server as a return value, whereas the message `boot` is sent to the local server in order to boot it. In the first case (message `local`), it is the return value of the operation that is of further use, while in the second case (message `boot`), it is the effect of the boot operation that matters.

5.4.2 Functions as Program Modules

Since functions are objects that can be stored in variables, it is easy to define and store any number of functions (i.e., miniature programs) and run them, whenever required, any number of times. Thus, defining functions and configuring their combinations can be a major part of programming in SuperCollider.

Figure 5.15 illustrates how to call a function that has been stored in a variable in various ways. The function `change_freq` in the example does two things: it calculates a new frequency for the sound by moving one semitone upward or downward from the previous pitch, and it sets the frequency of the sound to the new pitch.

```
s.boot;    // (boot Server before running example)
(
// Define a function and call it in different contexts
var synth;       // Synth creating the sound that is changed
var freq = 220; // frequency of the sound
var change_freq; // function that changes the frequency of the   so
und
var window;     // window holding buttons for changing the sound
var button1, button2, button3; // buttons changing the sound
// Create a synth that plays the sound to be controlled:
synth = {| freq = 220 | LFTri.ar([freq, freq * 2.01], 0, 0.1)}   .p
lay;

// Create frequency changing function and store it in variable chan
ge_freq
change_freq = { // start of function definition
   freq = freq * [0.9, 0.9.reciprocal].choose; // change freq   val
ue
   synth.set(\freq, freq);    // set synth's frequency to new   val
ue
}; // end of function definition

// Create 3 buttons that call the example function in various ways
window = Window("Buttons Archaic", Rect(400, 400, 340, 120));
//---------------Example 1---------------
button1 = Button(window, Rect(10, 10, 100, 100));
button1.states = [["I"]]; // set the label of button1
// button1 calls the function each time that it is pressed
button1.action = change_freq;    // make button1 change freq   once
//---------------Example 2---------------
button2 = Button(window, Rect(120, 10, 100, 100));
button2.states = [["III"]];
// Button2 creates a routine that calls the example function 3 time
s
button2.action = {  // make button2 change freq 3 times
   {3 do: {change_freq.value; 0.4.wait}} .fork; // play as   routin
e
```

```
};
//---------------Example 3---------------
button3 = Button(window, Rect(230, 10, 100, 100));
button3.states = [["VIII"]];
button3.action = {  // like example 2, but 8 times
    {8 do: {change_freq.value; 0.1.wait}} .fork; // play as   routi
ne
};
// use large size font for all buttons:
[button1, button2, button3] do: _.font_(Font("Times", 32));
// stop the sound when the window closes:
window.onClose = {synth.free};
window.front; // show the window
)
```

Figure 5.15
Multiple use of a function stored in a variable.

The code of the function consists of two statements:

```
{
  freq = freq * [0.9, 0.9.reciprocal].choose; // change freq   valu
e
  synth.set(\freq, freq);  // set synth's frequency to new value
}
```

This function is stored in the variable `change_freq` and then called in two different ways:

- It is stored in the action part of a GUI button so that when that button is pressed, it runs the function.
- It is called explicitly by a function inside a `Routine` that sends it the message value. (As noted in chapter 3, Routine has the ability to time the execution of its statements, and therefore it can run the function in question at timed intervals.)

5.4.3 Compilation and Evaluation: The Details

SuperCollider undergoes a three-step process every time that it executes code entered in a work space window: First, it compiles the code of the program and creates a function that can be evaluated. Second, SuperCollider evaluates that function. Finally, SuperCollider prints the result of the evaluation in the *post* window.

Figure 5.16 shows what happens when one runs the code `3 + 5`. The equivalent of the entire compilation plus the evaluation process can be expressed by the code `"3 + 5".`

interpret.

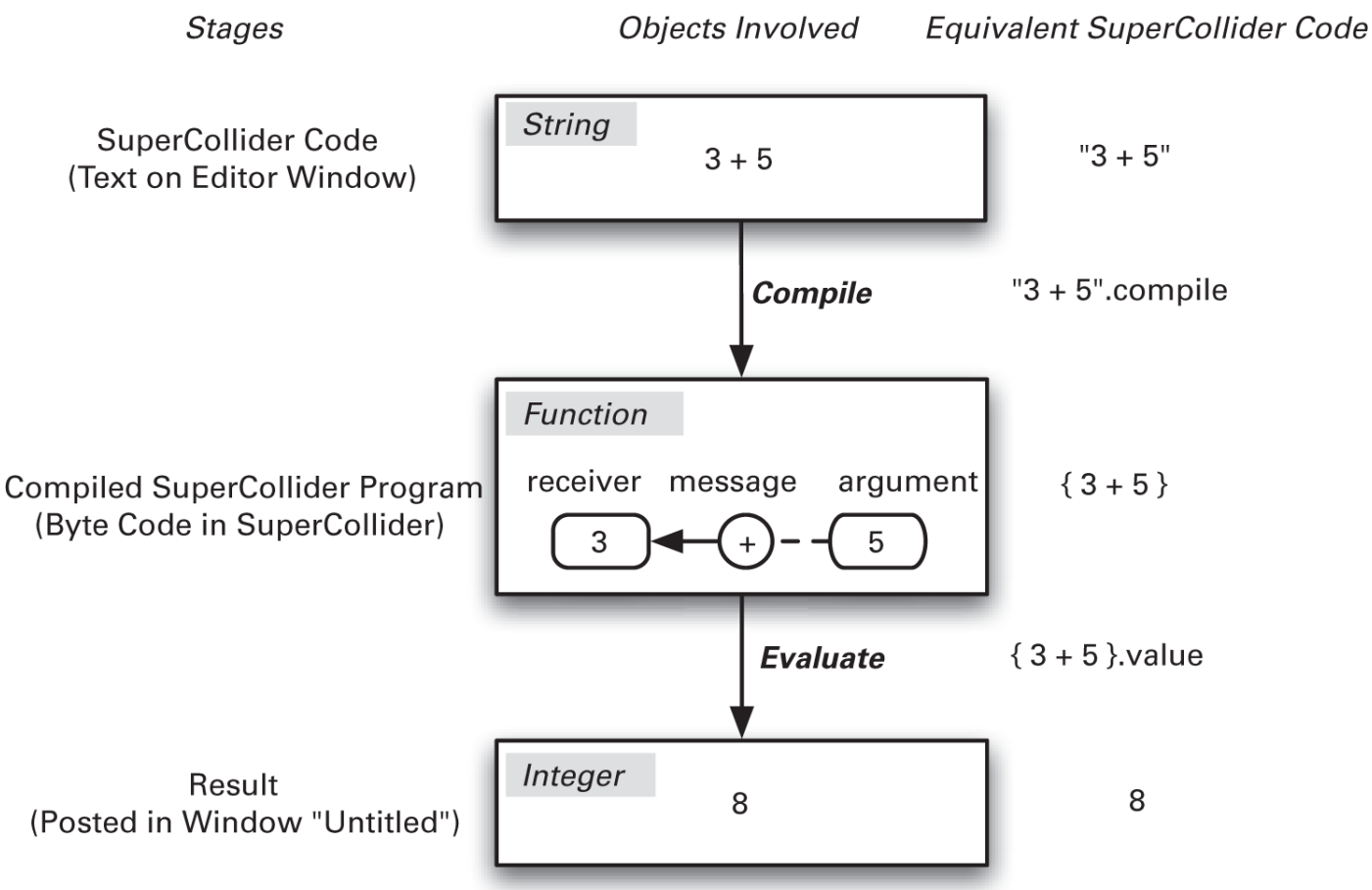


Figure 5.16
Compiling and evaluating code.

5.4.3.1 Who does the compiling? In SuperCollider, even the top-level processes of interaction with the user are defined in terms of objects inside the system. An easy way to see what happens is to cause an error and look at the error message and its call stack, which shows the sequence of methods called in reverse order from the most recent. For example, evaluate `1.error`. The bottom of the call stack shows the beginning of the compilation process:

```
Process:interpretPrintCmdLine  14A562F0
  arg this = <instance of Main>
```

Immediately above that is the next method call:

```
Interpreter:interpretPrintCmdLine  15055D00
  arg this = <instance of Interpreter>
```

This shows that the top-level object responsible for compiling and interpreting text input is an instance of class `Main`, and it delegates the interpretation to an instance of `Interpreter`, calling the method `interpretPrintCmdLine`.

5.4.3.2 Byte code: Looking at the compiled form of a function The compilation process consists in successively replacing the SuperCollider code of the program with pieces of byte code and data in the computer's memory. The compiler's task is first to parse (i.e., understand the program structure contained in the code) and then to translate that exact structure—including data and instructions—into byte code. To display the actual byte code of a compiled SuperCollider program, one sends the definition of the function representing the program the message `dumpByteCodes`. To obtain the definition of the function, one sends it the message `def`. Thus, to display the byte code of the example `3 + 5`, evaluate this line:

```
{3 + 5}.def.dumpByteCodes
```

Figure 5.17 explains the resulting printout.

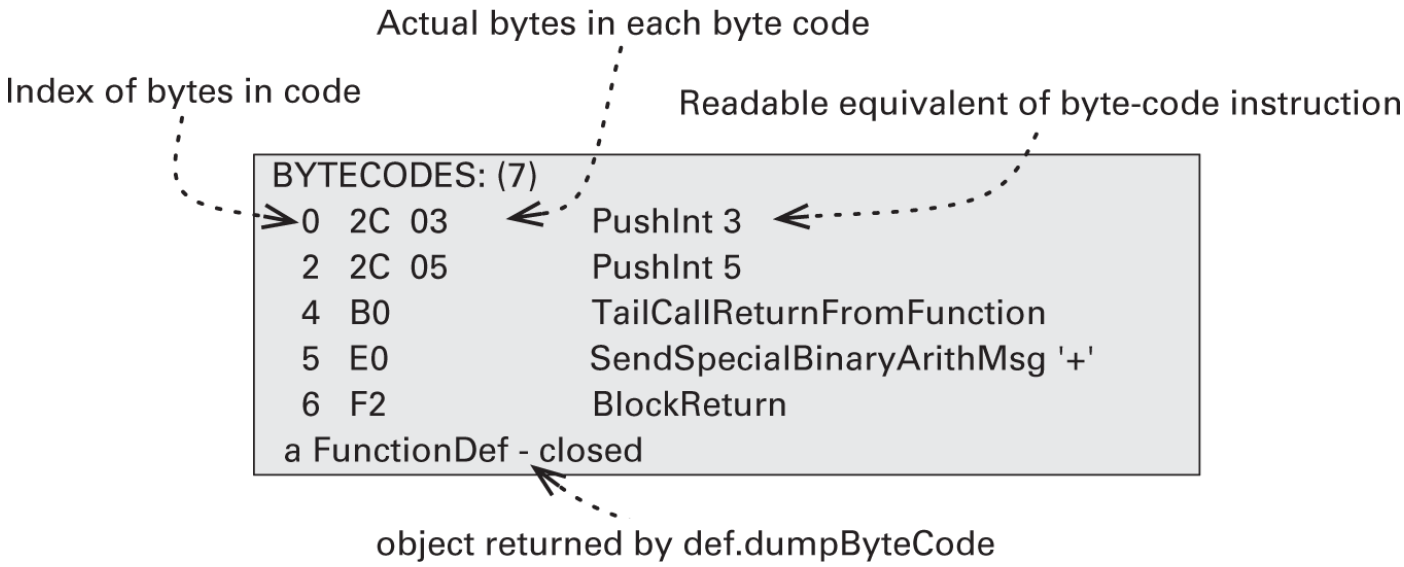


Figure 5.17
Byte code of the function `{3 + 5}`.

5.4.4 Functions with Arguments

Functions can have inputs for receiving data from the context that calls them. The inputs, if any, are defined at the beginning of the function, before any variables, and are called *arguments*. Arguments are variables of a function whose values can be set by the program that calls it. When a function needs to be evaluated with different sets of data each time, it defines as many arguments as there are data items required. The program

can then give data to a function by appending them as arguments in the value message. Here is how to define and call a function that computes and returns the sum of 2 numbers, *a* and *b*:

```
var sum2; // define variable to store the function;
// define the function and store it in variable sum2:
sum2 = {arg a, b;  // start of function definition,    arguments a,
b
  // the body of the function (the program) is here
  a + b  // compute and return the function of a and b
};  // end of function definition
// call the function, giving it the numbers 2 and 3 as arguments:
  sum2.value(2, 3);  // the returned value is 5
)
```

5.4.4.1 Defining arguments In SuperCollider, arguments are defined by prepending the declaration keyword `arg` or by enclosing them in vertical bars `| |`. (See figure 5.18.)

```
(
// a function that calculates the square of the mean of two numbers
var sq_mean;
sq_mean = {arg a, b;    // arguments a, b defined in arg   statemen
t form
   (a + b / 2).squared;
};
// calculate the square of the mean of 3 and 1:
sq_mean.value(3, 1);
)
```

Figure 5.18
Simple function with arguments.

Three dots (`...argName`) before the final (or only) argument name in the argument list can be used to collect any number of provided arguments into one array passed as a single argument to the function. (See figure 5.19.)

```
(
// a function that calculates the square of the mean of any numbers
var sq_mean_all;
sq_mean_all = {|. . .numbers | // using ellipsis and | |   argument
form
```

```
    (numbers.sum / numbers.size).squared;
};
// calculate the square of the mean of [1, 3, 5, −7]:
sq_mean_all.(1, 3, 5, −7); // short form: omit message 'value'
)
```

Figure 5.19
Using … for undefined number of arguments.

5.4.4.2 Default argument values The default values of arguments can be included in argument definitions in the same manner as variables. A default value is used only if no value was provided for the argument when the function was called. (See figure 5.20.)

```
(
var w_func;
w_func = {arg message = "warning!", bounds = Rect(200, 500,   500,
 100);
   var window;
   window = Window("message window", bounds).front;
   TextView(window, window.view.bounds.insetBy(10, 10))
        .string = message;
};
// provide text, use default bounds
w_func.(String.new.addAll(Array.new.addAll(" Major news! ")   .pyra
mid(7)));
)
```

Figure 5.20
Using and overriding default values of arguments.

Since functions are objects in SuperCollider—or, more exactly, "*first class objects*"[2]—their behavior can easily be extended to include other things besides running them with the message `value`. The following sections describe common ways of using functions.

5.4.5 Customizing the Behavior of Objects with Functions

Several classes of objects that deal with user interface, or with interactive features that should be easily set by the programmer, store functions in variables. Such functions in variables define how an object should react to certain messages. For example, buttons or other GUI widgets use the variable action to store the function that should be called when the user activates the widget by a mouse click.

In figure 5.21, the action of the button chooses between two messages to perform on the default Server, depending on the value (state) of the button. In figure 5.22, the action chooses between two functions, depending on the state of the button.

```
(
var window, button;
window = Window("Server Button", Rect(400, 400, 200, 200));
button = Button(window, Rect(5, 5, 190, 190));
button.states = [["boot!"], ["quit!"]];
button.action = {|me| Server.default perform: [\quit, \boot]   [me.
value]};
window.front;
)
```

Figure 5.21
Performing messages chosen by index.

```
(
var window, button;
window = Window("Server Button", Rect(400, 400, 200, 200));
button = Button(window, Rect(5, 5, 190, 190));
button.states = [["boot"], ["quit"]];
button.action = {| me |
    [{"QUITTING THE DEFAULT SERVER".postln;
         Server.default.quit;
    }, {"BOOTING THE DEFAULT SERVER".postln;
         Server.default.boot;
    }][me.value].value;
};
window.front;
)
```

Figure 5.22
Evaluating functions chosen by index.

5.4.6 Functions as Arguments in Messages for Asynchronous Communication

Asynchronous communication happens when a program requests an action from the system but cannot determine when that action will be completed. For example, it may ask for a file to be loaded or to be printed, but the time required for this to finish is unknown. In such a situation, it would be disruptive to pause the execution of the program while waiting for the action to complete. Instead, the program delegates the processing of the answer expected from the action to an independent process—

represented by a function—that waits in the background. Two common cases are described in the following sections.

5.4.6.1 Asynchronous communication with a server The system asks for an action to happen on a server, for example, to load a sound file into a buffer (`Buffer.read`). Since the time it will take the server to load the file is not known in advance, a function is given to `read` as an argument, which is executed when the server completes loading the buffer. Figure 5.23 demonstrates that the action passed as an argument to the read method is executed *after* the statement following Buffer.read. Note that the server must be running for this to work.

5.4.6.2 Dialogue windows Dialogue windows that demand input from the user employ an `action` argument to determine what to do when input is provided. This prevents the system from waiting indefinitely for the user:

```
s.boot;
(
Buffer.loadDialog(action: {|buffer|
  format("loaded % at: %", buffer, Main.elapsedTime).postln;
});
format("continuing at: %", Main.elapsedTime).postln;
)

s.boot // boot default server before running example
(
var buffer;
buffer = Buffer.read(Platform.resourceDir +/+ "sounds/a11wlk01    .w
av",
   action: {| buffer |
        format("loaded % at: %", buffer, Main.elapsedTime).postln;
   });
format("Reached this after 'Buffer.read' at: %", Main.   elapsedTim
e).postln;
buffer;
)
```

Figure 5.23
Asynchronous communication with a Server.

5.4.7 Iterating Functions

Iteration is the technique of repeating the same function a number of times. It may be run for a prescribed number of times (`anInteger.do(aFunction)`), an unlimited

number of times (`loop(aFunction)`) while a certain condition is true (`while`), or over the elements of a `Collection` (see section 5.6.3).

5.4.7.1 Iterating a specified number of times

`do`: Iterate n number of times, pass the count as argument:

```
10 do: {|i| [i, i.squared, i.isPrime].postln}
```

`!`: Iterate *n* number of times, pass the count as argument, collect results in an Array:

```
{10.rand * 3}!5
```

`for`: Iterate between a minimum and a maximum integer value:

```
for(30, 35, {|i| i.postln});
```

`forBy`: Iterate between two values, using a definable step:

```
forBy(2.0 10, 1.5, {|i| i.postln})
```

Note how different syntax options aid readability (e.g., `for(30, 35,…)` rather than `30.for(35,…)`.

5.4.7.2 Iterating while a condition is true The message `while` will repeatedly evaluate a function so long as a test function returns `true`: `{[true, false].choose}.while({"was true".postln;})`. It is usually coded like this:

```
(
var sum = 0;
while {sum = sum + exprand(0.1, 3); sum <10} {sum.postln}
)
```

5.4.7.3 Infinite (indefinite) loop Here, `loop` repeats a function until the process that contains the loop statement is stopped. It can be used only within a process that stops or pauses between statements; otherwise, it will hang the system with an infinite loop. (See figure 5.24.)

```
s.boot; // do this first
(    // then the rest of the program
var window, routine;
window = Window("close me to stop").front;
```

```
window.onClose = {routine.stop};
routine = {
    loop {
    (degree: -10 + 30.xrand, dur: 0.05, amp: 0.1.rand).play;
    0.05.rand.wait;
    }
}.fork;
)
```

Figure 5.24
`loop` and the use of Event—`(key:value).play`—to play notes.

5.4.8 Partial Application: Shortcut Syntax for Small Functions

It is possible to construct functions that apply arguments to a single message call by using the underscore character _ as a placeholder for an argument. For example, instead of writing `{arg x; x.isPrime}`, one can write `_.isPrime`. If more than one _ is included, then each _ takes the place of a subsequent argument in the function. Examples are shown in figure 5.25.

```
_.isPrime! 10
_.squared! 10
Array.rand(12, 0, 1000).clump(4) collect: Rect(*_)
(1..8).collect([\a, \b, _]);
(a: _, b: _, c: _, d: _, e: _).(*Array.rand(5, 0, 100));
```

Figure 5.25
Partial application.

5.4.9 Recursion

Recursion is a special form of iteration in which a function calls itself from inside its own code. To do this, the function refers to itself via the pseudovariable `thisFunction`. (A pseudovariable is a variable that is created and set by the system and is not declared anywhere in the SuperCollider class library. See section 5.3.6.2.) The value of `thisFunction` is always the function inside which `thisFunction` is accessed. Figures 5.26 and 5.27 show the difference in implementing the algorithm for computing the factorial of a number iteratively and using recursion. The recursive algorithm is shorter.

```
(
var iterative_factorial;
iterative_factorial = {|n|
```

```
    var factorial = 1;   // initialize factorial as factorial of 1
    // calculate factorial n times, updating its value each time
    n do: {|i| factorial = factorial * (i + 1)};
    factorial; // return the final value of factorial;
};
iterative_factorial.(10).postln;    // 10 factorial: 3628800
)
```

Figure 5.26
Iterative factorial.

```
// Define the factorial function and store it in variable f:
f = {|x| if (x > 1) {x * thisFunction.value(x-1)} {x}};
f.value(10);    // 10 factorial: 3628800
```

Figure 5.27
Recursive factorial.

Conciseness is not the only reason for using recursion. There are cases when only a recursive algorithm can be used. Such cases occur when one does not know in advance the structure and size of the data to be explored by the algorithm. An example is shown in figure 5.28.

```
(
/* a function that recursively prints all folders and files
found in a path and its subfolders */
{| path |
    // store function here for use inside the if's {}:
    var thisFunc = thisFunction;
    format("====== now exploring: %", path).postln;
    // for all items in the path:
    path.pathMatch do: {| p |
        // if the item is a folder, run this function on its conten
ts
        // otherwise print the file found
        if (p.last == $/) {thisFunc.(p ++ "*")} {p.postln}
    }
}.("*") // run function on home path of SuperCollider
)
```

Figure 5.28
Recursion over a tree of unknown structure.

5.4.10 Inspecting the Structure of a Function

A particular feature of functions is the ability to access a function's parts, which define its structure. An example is the following:

```
var foo;
foo = {|a = 1, b = 2| a.pow(b)};
foo.def.sourceCode.postln; // print sourceCode
```

The source code of a function is stored only if that function is *closed* (i.e., if it does not access the variables of an enclosing function). A function's `def` variable contains a `FunctionDef` object that also contains the names of the arguments and variables of the function and their default values. These are used by the SynthDef class, for example, to compile a function into a UGen graph and then into a SynthDef that can be used to create synths on the Server.

5.4.11 Scope of Variables in Functions

As mentioned in section 5.3, variables are accessible only within the context (i.e., the function) that defines them. However, if a function *mother_func* creates another function *child_func*, then *child_func* has access to the variables created within *mother_func*. This is useful when several functions want to share data. Thus, in figure 5.15, the variable `freq` is defined in the implicit top-level function, and the function stored in `change_freq` is a *child_func* that has access to this variable. The function `change_freq` can therefore both *read* (access) the value of the variable `freq` and set (*write*) it whenever it is called. The set of variables created by function *f* and made available to functions created within that function *f* is called a function's *closure*.

It is in fact possible to use a closure as a sort of simple and limited form of an object, where the variables defined in the top-level function of the closure serve as instance variables. The following two examples show how to model the behavior of the `Counter` class shown in section 5.5 without writing a class definition.

5.4.11.1 Modeling instances through functions that create other functions

Figure 5.29 defines a *mother_func* stored as `counter_maker`, which in turn creates and returns a *child_func*. Each time that `counter_maker` is run, it creates a new instance of its *child_func*. It also creates copies of its own variables—in this case, the argument variable `max_count` and the variable `current_count`, which are accessible only to its own child function.

```
(
// a function that creates a function that counts to any number
var counter_maker;
```

```
var window, button1, button2; // gui for testing the function

// the function that makes the counting function
counter_maker = {| max_count |
   // current_count is used by the function created below
   // to store the number of times that it has run
   var current_count = 0;
   {    // start of definition of the counting function
        if (current_count == max_count) {
              format("finished counting to %", max_count).postln;
              max_count;    // return max count for eventual use
        }{
              current_count = current_count + 1; // increment count
              format("counting % of %", current_count, max_count)
.postln;
              current_count    // return current count for eventual
use
              }
   }    // end of definition of the counting function
};

//-Test application for the counter_maker function-
// window displaying 2 buttons counting to different numbers
window = Window("Counters", Rect(400, 400, 200, 80));
// make a button for triggering the counting:
button1 = Button(window, Rect(10, 10, 180, 20));
button1.states = [["counting to 10"]];  // labels for button1
// make a function that counts to 10 and store it as action in butt
on1
button1.action = counter_maker.(10);
button2 = Button(window, Rect(10, 40, 180, 20));
button2.states = [["counting to 5"]];    // labels for button2
// make a function that counts to 5 and store it as action in butto
n2
button2.action = counter_maker.(5);
window.front;   // show the window
)
```

Figure 5.29
A function that creates functions that count.

So from one mother function, one can create multiple closures, each closure having its own set of variables and functions, and each function in that closure is able to run

multiple times. In this way, one can construct programs that make smaller programs that work on their own copies of data. In the present example, the function stored in `counter_maker` is run once with a `max_count` argument value of 10 and once with a `max_count` argument value of 5. Consequently, the first time, it creates a function that counts to 10; and the second time, one that counts to 5.

The effect of this technique is similar to defining instance variables, and the child functions that have access to these variables are similar to instances of a class that has access to these variables. Figure 5.30 shows how closures with their own variables are generated from a function.

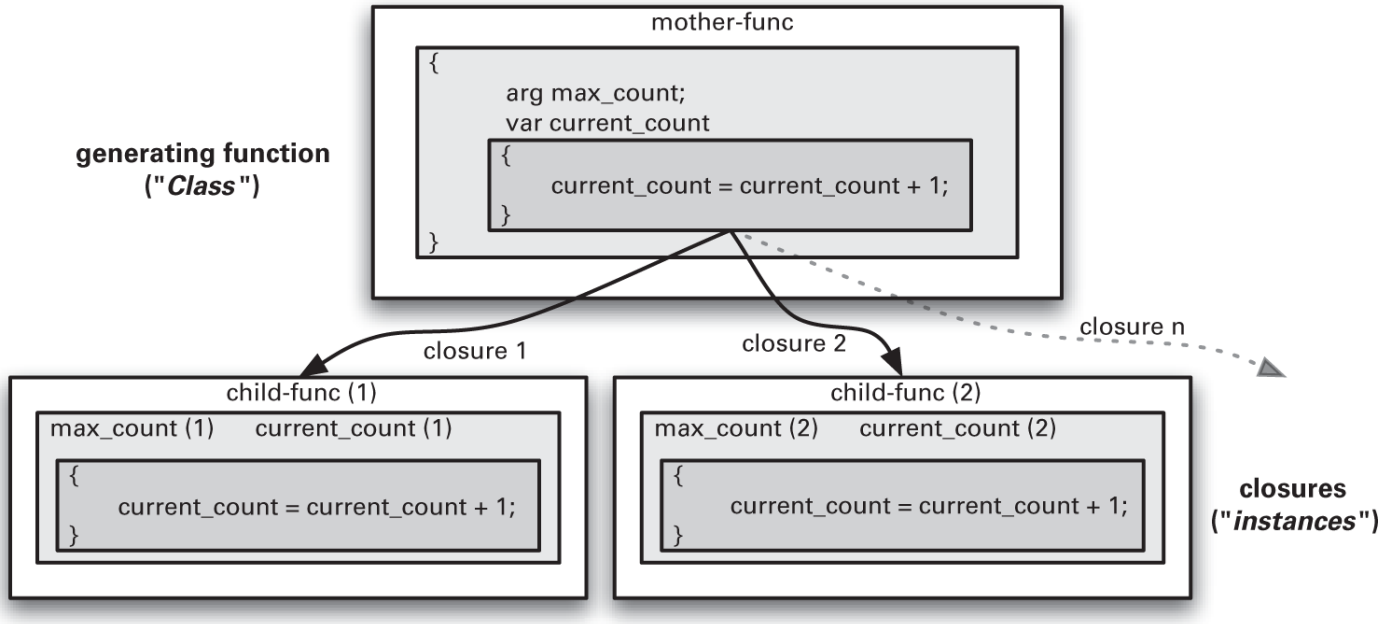


Figure 5.30
Functions created by functions as models of instances.

5.4.11.2 Functions in Events as methods This section extends the example from section 5.4.11.1 to add a further feature: the ability of each counter to reset itself. It also shows a more flexible technique for creating a graphical user interface: instead of a fixed number of counter items, a function is defined that can generate a GUI for any number of counters, whose maximum counts are given as arguments to the function. Instead of a function, the `counter_maker` in this example returns an `Event`, an object that can hold values associated to named keys. Instead of having a fixed, predefined number of instance variables, an Event can hold any number of key-value associations that function similarly to the named instance variables of an Object. In the current example the Event contains 3 keys—`'count1'`, `'reset_count'`, and `'max_count'`—whose values are bound to functions that operate on the variables of the `counter_maker` closure. These functions assigned as values to keys act the way an

instance method would in a normal class definition. Thus, an Event made by `counter_maker` is the model of an object with two variables and three methods. The code shown in figure 5.31 is hardly any bigger than the previous version, despite the addition of 2 features.

```
(
var counter_maker;         // creator of counters
var make_counters_gui;    // function making counters + a gui
/* a function that creates an event that counts to any number,
and resets: */
counter_maker = {| max_count |
   var current_count = 0;
   (   // the counter object is an event with 3 functions:
        count1: // function 1: increment count (stored as count1)
        {    // start of definition of the counting function
             if (current_count == max_count){
                 format("finished counting to %", max_count)   .po
stln;
             }{
                 current_count = current_count + 1; // increment
count
                 format("counting % of %", current_count,   max_co
unt).postln;
                 }
             },   // end of definition of the counting function
             reset_count: {// function 2: reset count (stored   as
reset_count)
                 format("resetting % counter", max_count).postln;
                 current_count = 0
             },
             max_count: {max_count} // function 3: return value
 of max_count
   )
};
// Function that makes several counters and a GUI to control them
make_counters_gui = {|. . .counts |
   var window, counter;
   window = Window("Counters",
             Rect(400, 400, 200, 50 * counts.size + 10));
   // enable automatic placement of new items in window:
   window.view.decorator = FlowLayout(window.view.bounds, 5@5,   5@
5);
```

```
    counts collect: counter_maker.(_) do: {| counter |
         Button(window, Rect(0, 0, 190, 20))
               .states_([["Counting to:" ++ counter.max_count   .asS
tring]])
               .action = {counter.count1};
         Button(window, Rect(0, 0, 190, 20))
               .states_([["Reset"]])
               .action = {counter.reset_count};
    };
    window.front;
};
make_counters_gui.(5, 10, 27); // example use of the GUI test   fun
ction
)
```

Figure 5.31
Functions stored in events as instance methods.

This example can be seen as a rudimentary class definition constructed without employing the regular class definition system of SuperCollider. It is left to the reader to extend the example in one further step, by storing the functions of `counter_maker` and `make_counters_gui` in an `Event` to model a class `Counter` with two class methods.

The syntax for running the functions stored in an Event is the same as that of a method call, (`receiver.message`), the only difference being that the first argument passed to a function in an Event is the Event itself. There is a catch, however. If one stores a function in an Event under the name of an instance method that is defined in the class Event, then that method will be run instead of the function stored by the user. So, for example, one cannot use a function stored in an Event under reset:

```
(reset: {"this is never called".postln;}).reset;
```

For more information on this approach, see chapter 8.

5.5 Program Flow Control and Design Patterns

Control structures are structures that permit you to choose the evaluation of a function depending on a condition. That is, a function is evaluated only if the value of a test condition is true. There are variants involving one or more functions. (For alternative syntax forms, see the Help file Syntax-Shortcuts.)

5.5.1 If Statements

Run a function only if a condition is true:

```
if ([true, false].choose) {"was true".postln}
```

Run a function if a condition is true; otherwise, run another function:

```
if ([true, false].choose) {"was true".postln} {"was false".   postl
n}
```

5.5.2 Case Statements

A `case` statement is a sequence of function pairs of the form "condition-action." The condition functions are evaluated in sequence until one of them returns true. Then the action function is evaluated and the rest of the pairs are ignored. One can add a single default action function at the end of the pairs sequence, which will be executed if none of the condition functions returns true.

```
(
i = [0, 1, inf].choose;
x = case {i == 0} {\no}
  {i == 1} {\yes}
  {\infinity};
)
```

5.5.3 Switch Statements

A `switch` statement matches a given value to a series of alternatives by checking for equality. If a match is found, the function corresponding to that match is evaluated. The form of the switch statement is similar to that of the case statement. The difference is that the switch statement uses a fixed test—that of equality with a given value—whereas the case statement uses a series of independent functions as tests.

```
(
switch ([0, 1, inf].choose,
  0, {\no},
  1, {\yes},
  {\infinity})
)
```

5.5.4 Other Control Techniques: Behavior Patterns

Selecting among alternatives for directing the execution flow of a program is not limited to the statements above. There are many techniques addressing this topic, some of which are also known as *Design Patterns* (Gamma et al. 1994; Beck, 1996). Typically,

techniques in this category would fall under the group *Behavior Patterns*. Examples of such patterns are *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Observer,* and *State.* Beck (1996) classifies behavior patterns into two major categories. Under "Method," he lists patterns that are based on the organization of an algorithm inside methods. Under "Message," he classifies patterns that use message passing to create algorithms. These patterns can be very small but equally powerful. An example is the *Choosing Message* pattern (Beck, 1996, pp. 45–47). Instead of choosing among a number of alternatives with an if statement or a switch statement, one delegates the choice to the methods of the possible objects involved. For example, consider an object that represents an entry in a list of publications, and that responds to the message responsible by returning some object that represents the name of the person who is responsible for the object. For film publications, the "responsible" person is the producer, for edited books, it is the editor, and for single-author books, it is the author. The *Choosing Message* pattern says that instead of writing

```
responsible {|entry|
  case {entry.isKindOf(Film)} {^entry.producer}
    {entry.isKindOf(EditedBook)} {^entry.editor}
    {^entry.author}  // in all other cases, return the author
},
```

one writes

```
responsible {|entry|^entry.responsible}
```

and then codes the different reactions to responsible in the classes of the objects that are involved:

```
// add method "responsible" in 3 previously defined classes:
+ Publication {responsible {^author}}
+ Film {responsible {^producer}}
+ EditedBook {responsible {^editor}}
```

In this example, Publication is the default class for entries and gives the default method; all other classes for entries are subclasses of Publication. Only those classes which deviate from the default responsible method are needed to redefine it. (See section 5.2.1.2 for the syntax of methods and class extensions.)

The power of this technique is first, that the number of choices can easily be extended by creating new classes; and second, that the method responsible for each class can be as complex as needed, without resulting in a huge case statement that aggregates all the

choices for “responsible” in one place. In other words, complexity is reduced—or, rather, broken into pieces in an elegant way—by delegating responsibility for different parts of the algorithm to different classes. Thus, algorithms are organized by the combination of a number of method calls, which split the algorithm into pieces and delegate the responsibility for different parts of the algorithm to different classes. As a result, methods tend to contain very little code, often just a single line. Although this may seem confusing at the first encounter, it gets clearer as one becomes familiar with the style of code that pervades good object-oriented programming.

5.6 Collections

Collections are objects that hold a variable number of other objects. For example, figure 5.32 shows a program that adds a new number to a sequence each time the user clicks on a button, and then plays the sequence as a “melody.”

```
s.boot;     // boot the server first;
(
var degrees, window, button;
window = Window("melodies?", Rect(400, 400, 200, 200));
button = Button(window, window.view.bounds.insetBy(10, 10));
button.states = [["click me to add a note"]];
button.action = {
   degrees = degrees add: 0.rrand(15);
   Pbind(\degree, Pseq(degrees), \dur, Prand([0.1, 0.2, 0.4],   in
f)).play;
};
window.front;
)
```

Figure 5.32
Building an Array with add.

The above example builds a sequence of notes by adding a new random integer between 0 and 15 each time. Note that adding an element to nil creates an array with the added element (i.e., `nil add: 1` results in `[1]`).

The subclass tree of `Collection` is extensive (`Collection.dumpClassSubtree`) and is summarized in the Help file Collections. Collections can be classified into three types according to the way in which their elements are accessed.

Collections whose elements are accessed by numeric index. For example, `[0, 5, 9]. at(0)` accesses the first element of the array `[0, 5, 9]`, and `[0, 5, 9].put(1, \hello)` puts the symbol `\hello` into the second position of array `[0, 5, 9]`. Such

collections include `Array`, `List`, `Interval`, `Range`, `Array2D`, `Signal`, `Wavetable`, and `String`. Numeric indices in SuperCollider start at 0; that is, 0 refers to the first element in a collection. Accessing an element at an index past the size of the collection returns `nil`. Other messages for access exist—`wrapAt`, `clipAt`, `foldAt`—that modify invalid index numbers so they always return some element. There are collections that hold only a specific kind of object, such as `Char` (`String`), `Symbol` (`SymbolArray`), or `Float` (`Signal`, `Wavetable`).

Collections whose elements are accessed by using a symbol, or another object, as the index. For example, `(a: 1, b: 2)[\a]` returns `1`. Such collections are `Dictionary`, `IdentityDictionary`, `MultiLevelIdentityDictionary`, `Library` (a global, nested `MultiLevelIdentityDictionary` that can be accessed by series of objects as indices), `Environment`, and `Event`. All such collections are made up of `Association` objects, which are pairs that associate a key to a value and are written as `key->value`. Although it is possible to look up such pairs both by key and by value, dictionaries are optimized for lookup by key.

Collections whose elements are accessed by searching for a match to a condition. For example, look at `Set[1, 2, 3, 4, 5] select: (-> 2)`. These are `Set` and `Bag`.

5.6.1 Creating Collections

The generic rule for creating a collection is to enclose its elements in brackets `[]`, separating each element by a comma. If the class of a collection is other than Array, it is indicated before the brackets:

```
List[1, 2, 3]; LinkedList[1, 2, 3]; Signal[1, 2, 3]; Dictionary[\a->1, 2->pi, \c-> 'alpha']; Set[1, 2, 3]
```

In addition, there are several alternative techniques for notating and generating specific types of collections:

- An arithmetic series can be abbreviated by giving the beginning and end values and, optionally, the step between subsequent values: `(1..5); (1, 1.2 .. 5)`.
- An `Event` can be written as a pair of parentheses enclosing a list of the associations of the Event written as keyword-value pairs: `(a: 1, b: 2)`.
- Environments and Events can be created from functions with the message `make`. (See section 5.6.8.)

There are several messages for constructing numerical Arrays algorithmically. For example:

```
Array.series(5, 3, 1.5); Array.geom(3, 4, 5); Array.rand(5,   -10,
 10)
```

Wavetables and Signals are raw Arrays of floating-point numbers that can be created from functions such as sine or Chebyshev polynomials, or window shapes such as Welch.

The class `Harmonics` constructs Arrays that can be used as wavetables for playing sounds with the UGen `Osc` and its relatives.

5.6.2 Binary Operators on Collections

Most binary operators on collections can work both between two collections of any size and between a collection and a noncollection object: `(0..6) < (3..0); (0..6) + (3..0); 10 * (1..3); (2..5) + 0.1`. One can append an adverb to a binary operator to specify the manner in which the elements of two collections are paired for the operation. For example:

```
[10, 20, 30, 40, 50] + [1, 2, 3] // default: shorter array   wraps
[10, 20, 30, 40, 50] +.s [1, 2, 3] // s = short. operate on   short
er array
[10, 20, 30, 40, 50] +.f [1, 2, 3] // f = fold. Use folded   indexi
ng
```

5.6.3 Iterating over Collections

The following messages iterate over each element of a collection with a function:

`do(function)`: Evaluate function over each element; return the receiver.

```
(1..5) do: _.postln
```

`collect(function)`: Evaluate function over each element; return the collected results of each evaluation.

```
(1..5) collect: _.sqrt
```

`pairsDo(function)`: Iterate over adjacent pairs of elements of a collection.

`inject(function)`: Iterate passing the result of each iteration to the next one as an argument:

`keysDo, keysValuesDo, associationsDo, pairsDo, keysValuesChange`: These work on dictionaries as follows:

```
(a: 10, b: 20) keysDo: {|key, index| [key, index].postln}
(a: 10, b: 20) keysValuesDo: {|k, v, i| [k, v, i].postln}
(a: 10, b: 20) associationsDo: {|assoc, index| [assoc, index].   po
stln}
(a: 10, b: 20) pairsDo: {|k, v, i| [k, v, i].postln}
(a: 10, b: 20) keysValuesChange: {|key, value, index| value   + ind
ex}
```

5.6.4 Searching in Collections

The following messages search for matches and return either a subset or a single element from a collection:

- `select(foo)`: Return those elements for which foo returns true: `(1..5) select: (_ > 2).`
- `reject(foo)`: Return those elements for which foo returns false: `(1..5) reject: (_ > 2).`
- `detect(foo)`: Return the first element for which foo returns true: `"asdfg" detect: {|c| c.ascii > 100}.`
- `indexOf(obj)`: Return the index of the first element that matches obj: `"asdfg" indexOf: $f.`
- `includes(obj)`: Return true if the receiver includes obj in its elements: `"asdfg" includes: $f.`
- `matchRegexp(string, start, end)`: Perform matching of regular expressions on a string.

5.6.5 Restructuring Collections

A full account of the structure-manipulation features of the SuperCollider language would require a chapter of its own. For full details, the reader is referred to the Help files of the various Collection classes (and those of their superclasses, such as `Collection` and `SequenceableCollection`). Here are a few examples of some of the more commonly used methods:

- `reverse`: Reverse the order of the elements: `(1..5).reverse.`
- `flop`: Turn rows into columns in a 2D collection: `[[1, 2], [\a, \b]].flop.`
- `scramble`: Rearrange the elements in random order: `(1..5).scramble.`
- `clump(n)`: Create subcollections of size *n*: `(1..10).clump(3).`
- `stutter(n)`: Repeat each element *n* times: `(1..5).stutter(3).`
- `pyramid(n)`, where `1 <= n <= 10`: Rearrange in quasi-repetitive patterns. `(1..5).pyramid(5).`

• `sort(foo)`: Sort using foo as the sorting function. Default sorts in ascending order: `"asdfg".sort`. Descending order is specified like this: `"asdfg" sort: {|a, b| a > b}`.

Further powerful restructuring, combinatorial, and search capabilities are discussed in the Help files J Concepts in SC and List Comprehensions.

5.6.6 IdentityDictionary

`IdentityDictionary` is a dictionary that retrieves its values by looking for a key that is identical to a given index. "Identical" means that the key should be the same object as the index. For example, the two strings `"hello"` and `"hello"` are equal but not identical:

```
"hello" == "hello"; // true: the two strings are equal
"hello" === "hello"; // false: the two strings are not identical
```

By contrast, symbols that are written with the same characters are always stored as one object by the compiler and are therefore identical: `\hello === \hello` returns `true`. Thus:

```
a = IdentityDictionary["foo"->1]; // store 1 under the "foo" as key
a["foo"]; // nil!
```

The second `"foo"` is not identical to the first one.
However:

```
a = IdentityDictionary[\foo->1];
a[\foo]; // Returns 1
```

Searching for a matching object by identity is much faster than searching by equality. Therefore, an `IdentityDictionary` is optimized for speed. It serves as a superclass for `Environment`, which is the basis for defining Environment variables. Accessing an Environment variable thus means looking it up by identity match. Although this is a fast process, it is still considerably more expensive for computing cycles than accessing a "real" variable!

`IdentityDictionary` defines two instance variables: `proto` and `parent`. These are used by the `Environment` and `Event` classes to provide a default environment when needed (see section 5.6.8). The parent scheme makes it possible to build hierarchies of parent events in a way similar to that for class hierarchies.

5.6.7 Environment

An `Environment` is an `IdentityDictionary` that can evaluate functions which contain environment variables (see section 5.5). To make an Environment from a function, use the message `make`:

```
Environment make: {~a = 10; ~b = 1 + pi * 7.rand;}
```

This is not just a convenient notation; it also allows one to compute variables that are dependent on the value of variables previously created in the Environment:

```
Environment make: {~a = pi + 10.rand; ~b = ~a pow: 5}
```

The message `use` evaluates a function within an Environment.

```
Environment make: {~c = 3} use: {~a = 2 pow: 10.rand; ~c + ~a}
```

`Environment.use(f)` evaluates `f` in an empty Environment:

```
Environment use: {~a = 10; ~b = 1 + pi * 7.rand; ~c}
```

In addition, an `Environment` can supply values from its variables to the arguments of a function that is evaluated in it with the message `valueEnvir`. Only values for those arguments that are not provided by `valueEnvir` are supplied:

```
(a: 1, b: 2).use({~a + ~b}); // using Environment variables
```

Supplying arguments to a function from the Environment with `valueEnvir`:

```
(a: 1, b: 2).use({{|a, b|a + b} .valueEnvir(3)})
```

Note that the function must be explicitly evaluated with `valueEnvir` for this to work. Therefore, the following is not the right way to supply arguments with `use`:

```
(a: 1, b: 2).use({|a, b| a + b})
```

Here, `valueEnvir` in normal code text outside of `use` draws on the `currentEnvironment`:

```
~a = 3; ~b = 5;
{|a, b| a + b}.valueEnvir
```

Patterns make up a specific extension library within SuperCollider that is very useful for musical scheduling, and they rest on the Environment mechanisms to manipulate musical data. (They actually use `Event`, a subclass of Environment, which we will introduce next.) Patterns exploit the ability to supply values for arguments from an Environment with `valueEnvir` when playing instruments that are defined as functions.

5.6.8 Event

`Event` is a subclass of `Environment` with several additional features. An Event itself is playable:

```
(degree: 2, dur: 3).play
```

Event stores several prototype Events in its class variables that embody default musical event types (a class variable is globally available to the instances of its class and its subclasses). These Events define a complete musical environment, covering aspects such as tuning, scales, legato, chords and chord strumming, MIDI, and playing with different instruments. To play, an Event receives or selects a parent Event as its Environment and overrides only those items of the parent that deviate from the default settings.

For example, the parent Event of `(degree: 5)` is `nil` before playing: `(degree: 5).parent`. To run `(degree: 5).play`, the Event sets its own parent Event, which can be printed by `(degree: 5).play.parent.asCompileString`. The parameters of this Environment also compute and set the final parameters that are needed to play the Event. In the present example, these are `freq`, `amp`, and `sustain`, as can be seen in the resulting Event:

```
s.boot;  // boot the server first. Run each following line   separa
tely:
(degree: 5).parent; // the parent before playing is nil
(degree: 5).play.parent.asCompileString; // The parent has   been s
et
(degree: 5).play; // Event, becomes ('degree': 5, 'freq':   440,. .
.)
```

5.7 Working with Classes

Classes are the heart of the SuperCollider system because they define the structure and behavior of all objects. All class definitions are contained in the SCClassLibrary folder (perhaps within the app bundle) or in the platform-specific extension folder, in files with the extension .sc. By studying these definitions, one can understand the function of

any part of the system in depth. By writing one's own classes or modifying existing classes, one can extend the functionality of the system.

5.7.1 Encapsulation, Inheritance, and Polymorphism

The three defining principles of object-oriented languages are *Encapsulation*, *Inheritance*, and *Polymorphism*. *Encapsulation* means that data inside an object are accessible only to methods that belong to that object. This protects the data of the object from external changes, thereby aiding in creating consistent and safe programs. In *Polymorphism*, the same message can correspond to different behaviors according to the class of the object that receives it. In section 5.5.4, an entry of class `Film` responds differently to the message `responsible` than an entry of class `EditedBook` does. *Inheritance*, on the other hand, entails that any subclass of `Publication` that does not define its own method responsible will use the method as defined in `Publication` instead. (See also section 5.7.4.) Together, these three features are responsible for the capabilities of object-oriented languages.

5.7.2 Compiling the SuperCollider Class Library

In contrast to code executed from a window holding SuperCollider code, which can be run at any time, changes made in class definition code take effect only after recompiling the SuperCollider Class Library. (See the Shortcuts Help file for platform-specific info on how to do this.) Compiling the library rebuilds all classes and resets the entire memory of the system.

5.7.3 Defining a Class

The structure of a class is defined by its variables and its methods. Additionally, a class may define class variables, constants, and class methods.

As noted above, class variables are accessible by the class itself, as well as by all instances, whereas instance variables are accessible only inside methods of the instance in question. Constants are like class variables, except that their values are set at the definition statement and cannot be changed subsequently. For example, the class Char defines several constants that hold the unprintable characters for a new line, form feed, tab, and space, as well as the character comma.

Class methods are addressed to the class, and instance methods to instances of that class. For example, in `Window.new("test," Rect(500, 500, 100, 100)).front`, the class method `new` is addressed to the class `Window` and returns an appropriate window for the platform on which SuperCollider is running, and the method `front` is addressed to the instance created by the method `new`.

A class may inherit variables and methods from another class, which is called its *superclass*. Inheritance works upward over a chain of superclasses, and always up to

the superclass of all classes: `Object`. Before explaining the role and syntax of each element in detail, here is an example showing the main parts (figure 5.33).

As seen in figure 5.33, the code that defines a class has two major characteristics in common with that of a function: it is enclosed in `{}`, and it starts with variable declarations followed by program code. The code of a class definition is organized in two sections: variable declarations and method definitions. (No program statements may be included in the definition of a class other than those contained in variable declarations and methods.) Class syntax is summarized next.

The name of the class is indicated at the start of the definition. If the class has a superclass other than Object, it is indicated like this:

```
Integer: SimpleNumber {// define Integer as subclass of SimpleNumbe
r. . .
```

In addition to `var` statements that declare instance variables, there can be `classvar` statements that declare class variables and `const` statements that create constants. For example, the class `Server` has a class variable `set` that stores all servers known to the system. One can quit these servers with `Server.set do: _.quit`.

The special signs < and > prepended to a variable name in a variable declaration statement construct corresponding methods for getting or setting the value of that variable:

```
var <freq; // constructs method: freq {^freq};
var >freq: // constructs method: freq_ {|argFreq| freq =   argFreq}
```

For example, the class definition `Thing {var <>x;}` is equivalent to

```
Thing {var x;
  x {^x}
  x_ {arg z; x = z;}
}
```

The declaration of any variables of a class is followed by the definitions of its methods. A method is defined by the name of the method, followed by the definition of the function that is executed by that method.

The sign `*` before a method's name creates a class method:

```
*new {arg x=0, y=0; ^super.newCopyArgs(x, y);} // (from class   Poi
nt)
```

The default return value of an instance method is the instance that is executing that method (i.e., the receiver of the message that triggered the method). To return a different value, one writes the sign ^ before the statement whose value will be returned. The method `freq {^freq}` returns the value of the variable freq. The sign ^ also has the effect of *returning* from the function of the method, which means that any further statements will not be executed. This effect can be useful:

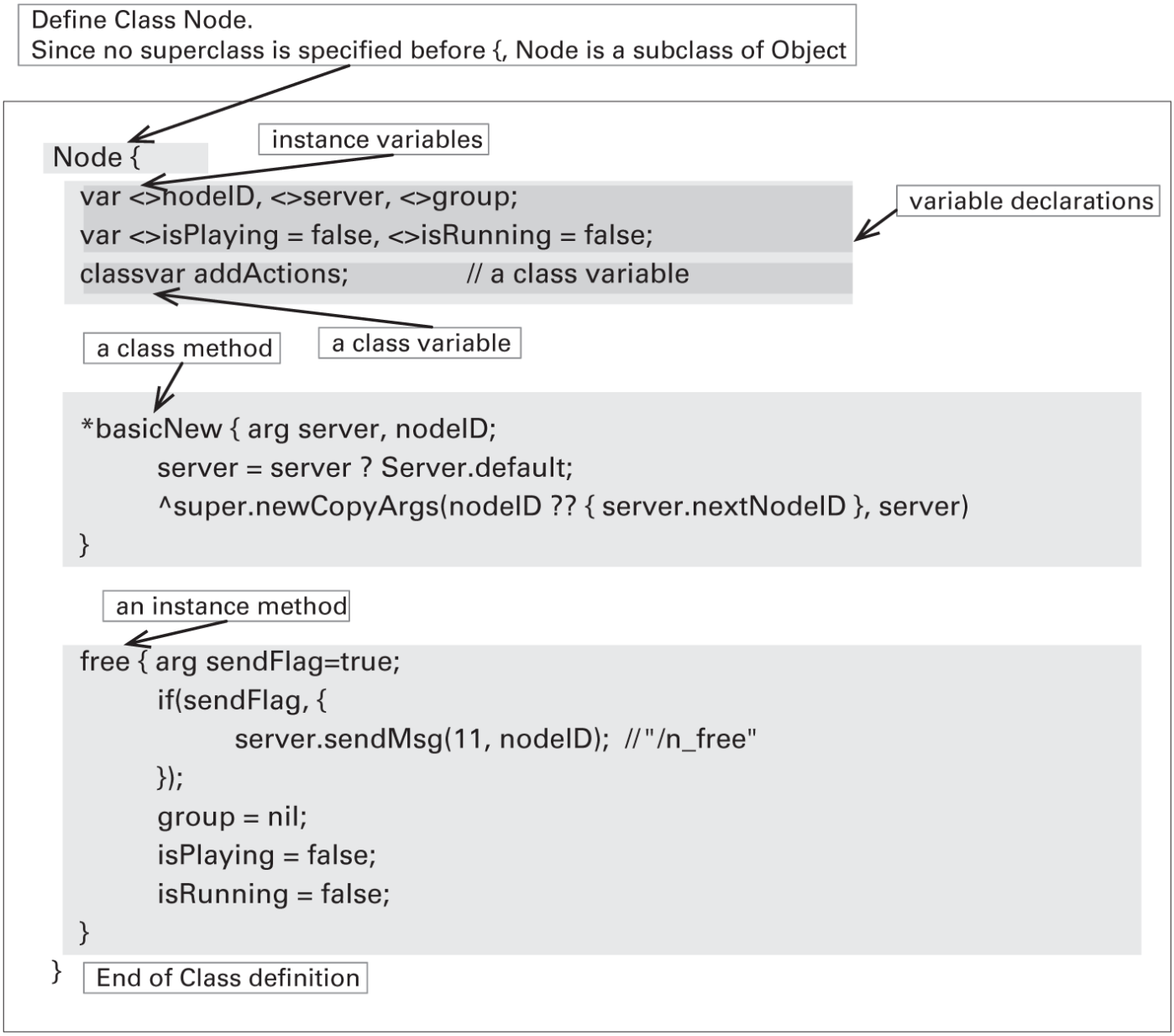


Figure 5.33
Summary of class definition parts (excerpt from the definition of a class node).

```
count1 {
  if (current_count >= max_count) {^current_count};
  // the next statement is executed only if current_count > max_cou
```

```
nt:
  ^current_count = current_count + 1;
}
```

Identifiers starting with an underscore (_) inside methods call *primitives*; that is, computations that are undertaken by compiled code in the system, and whose code can be seen only in the underlying source code of the SuperCollider application. A primitive returns a value if it can be called successfully. Otherwise, execution continues to the next statement of the method's code:

```
*newCopyArgs {arg. . .args; // (from class Object)
   _BasicNewCopyArgsToInstVars
   ^this.primitiveFailed // this is called only if BasicNew   CopyA
rgsToInstVars fails
}
```

Three special keywords can be used in methods: `this` refers to the object that is running the method (the *receiver*); `thisMethod` refers to the method that is running; `super` followed by a message looks up and evaluates the method of the message in the superclass of the instance that is running the method.

If the class method `*initClass` is defined, then it will be run right after the system is compiled. It is used to initialize any data needed. To indicate that a class needs to be initialized *before* the present initClass is run, one includes the following code in the definition of initClass:

```
Class.initClassTree(NameOfClassToBeInitialized);
```

A class is usually defined in one file. If the same class name is found in definitions in two or more files, then the compiler issues the message duplicate class found, followed by the name of the duplicate class. However, one can extend or modify a class by adding or overwriting methods in a separate file. The syntax for adding methods to an existing class is

```
+Function {// + indicates this extends an existing class
   // the code of any methods comes here
   update {. . . args | // method update
        this.valueArray(args);
   }   // other methods can follow here
}
```

5.7.4 Inheritance

A class may inherit the properties of another class. This principle of inheritance helps organize program code by grouping common shared properties of objects in one class, and by defining subclasses to differentiate the properties and behaviors of objects that have a more specialized character. For example, the class `Integer` inherits the properties of the class `SimpleNumber`. SimpleNumber is called the *superclass* of Integer, and Integer is called a *subclass* of SimpleNumber. `Float`, the class describing floating-point numbers such as 0.1, is also a subclass of SimpleNumber. Classes are thus organized into a family tree. The following expression prints out the complete SuperCollider class tree: `Object.dumpClassSubtree`.

5.7.5 Metaclasses

Since all entities in SuperCollider are objects, classes are themselves objects. Each class is the sole instance of its *metaclass*. For example, the class of `Integer` is `Meta_Integer`, and consequently, Integer is the only instance of the class Meta_Integer. All metaclasses are instances of `Class`. The following examples trace the successive classes of objects starting from the Integer 1 and going up to `Class` as the class of all Metaclasses.

The cycle `Class-Meta_Class-Class` in figure 5.34 shows the end of the Class relationship tree. Since the class of Class is Meta_Class and Meta_Class is also a Class, those two classes are the only objects that are instances of one another:

```
Class.class  // the class of Class is Meta_Class
Meta_Class.class  // the class of Meta_Class is Class

1.class    // the class of Integer 1: Integer
1.class.class        // the Class of the Class of Integer 1:   Meta
_Integer
// the Class of the Class of the Class of Integer 1:
1.class.class.class  // Class
// the Class of the Class of the Class of the Class of Integer 1
1.class.class.class.class    // Meta_Class
// the Class of the Class of the Class of the Class of the Class of
1
1.class.class.class.class.class // Class
Class.class            // the Class of Class is Meta_Class
Meta_Class.class     // the Class of Meta_Class is Class
```

Figure 5.34
Classes of classes.

Class methods are equivalent to instance methods of the class's Metaclass. For instance, the class method `*new` of Server is an instance method of Meta_Server. (See

figure 5.35.)

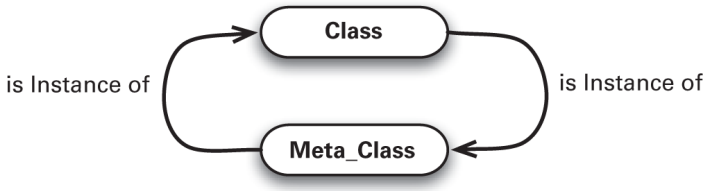


Figure 5.35
`Class` and `Meta_Class` are mutual instances of each other.

5.7.6 The SuperCollider Class Tree

At the top of the class hierarchy of SuperCollider is the class `Object`. This means all other classes inherit from class Object as its subclasses, and consequently all objects in SuperCollider share the characteristics and behavior defined in class Object. Object defines such global behaviors as how to create an instance, how an object should react to a message that is not understood, how to print the representation of an object as text, and so on. Any subclass can override this default behavior in its own code, as well as extend it by defining new variables and methods. The tree formed by Object and its subclasses thus describes all classes in the SuperCollider system.

5.7.7 Notifying Objects of Changes: Observer and Adapter/Controller Patterns

This section shows how to convert the class model from earlier in the chapter into a *real* class. The Observer design pattern implemented in the class Object allows one to attach objects (called *dependants*) to any object in such a way that they are updated when that object notifies itself with the message `changed` and an optional list of arguments. This results in the message `update` (along with the arguments) being sent to each of the dependants. It is the responsibility of the dependants to know how to respond correctly to an update message, and the changing object can remain ignorant of the kind and number of its dependants. Thus, it is possible to attach a sound, a GUI element, or any other object or process to another object and make it respond to changes of that object in any manner. Crucially, this can happen without having to modify the class definition of the changing object to deal with the specifics of the objects being notified. This technique is similar to the design pattern known as *Model-View-Controller (MVC)*. The goal of this pattern is to separate data or processes (the model) from their display (views) and from the control mechanisms, so as to permit multiple displays across different media and platforms.

The present example adds auditory displays and a GUI display that respond to counter changes. These displays are completely independent from each other and from the counter in both code and functionality, in the sense that one can attach a display or

remove it from any counter at any moment, and one can attach any number of displays to one counter.

The definition of the Counter class is shown in figure 5.36.

```
Counter {
    // variables: maximum count, current count
    var <>max_count, <>current_count = 1;
    // class method for creating a new instance
    *new {| max_count = 10 |
        ^super.new.max_count_(max_count)
    }
    // if maximum count not reached, increment count by 1
    count1 {
        if (current_count >= max_count) {
            this.changed(\max_reached)
        }{
            current_count = current_count + 1;
            this.changed(\count, current_count);
        }
    }
    // reset count
    reset {
        current_count = 1;
        this.changed(\reset);
    }
}
```

Figure 5.36
Counter class.

This must be placed in a file called Counter.sc in the SCClassLibrary folder and SC then recompiled (e.g., with "Recompile Class Library" under the Language menu). After that, boot the server and add the SynthDefs for the sounds (see figure 5.37).

```
s.boot;
(
SynthDef("ping", {| freq = 440 |
    Out.ar(0,
        SinOsc.ar(freq, 0,
            EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)
    ))
}).add;
```

```
SynthDef("wham", {
   Out.ar(0, BrownNoise.ar(
        EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)
   ))
}).add;
)
```

Figure 5.37
SynthDefs for the Counter model example.

Next, create five counters and store them in `~counters`:

```
~counters = (6, 11 .. 26) collect: Counter.new(_);
```

Now create a sound adapter to follow changes in any counter it is added to (see figure 5.38).

```
(
~sound_adapter = {| counter, what, count |
   switch (what,
        \reset, {Synth("wham");},
        \max_reached, {counter.reset},
        \count, {Synth("ping",
              [\freq, count.postln * 10 + counter.max_count * 20]
              )
        }
   )
};
)
```

Figure 5.38
A dependant that plays sounds.

The `sound_adapter` function receives `update` messages from a `Counter` object and translates them into actions according to the further arguments of the message. In this sense, it is similar to an Adapter pattern. (An Adapter pattern translates between incompatible interfaces.)

This works because the class `Function` defines the method `update` as a synonym for value, thus conveniently allowing it to work as a dependant in a straightforward manner:

```
update {|obj, what. . .args| this.value(obj, what, *args)}
```

Attach the sound adapter to all five counters:

```
~counters do: _.addDependant(~sound_adapter);
```

Then start a routine that increments the counters at 0.25-second intervals:

```
~count = {loop {~counters do: _.count1; 0.25.wait}}.fork;
```

The routine can be stopped with `~count.stop`. But before doing that, let's add GUI displays for the counters. (See figure 5.39.)

```
(
~make_display = {| counter |
   var window, label, adapter, stagger;
   window = Window(
        "counting to" ++ counter.max_count.asString,
        Rect(stagger = UniqueID.next % 20 * 20 + 400, stagger,   20
0, 50)
   );
   label = StaticText(window, window.view.bounds.insetBy(10, 10));
   adapter = {| counter, what, count |
               {label.string = counter.current_count.asString}   .d
efer
   };
   counter addDependant: adapter;
   /* remove the adapter when window closes to prevent error in
   updating non-existent views: */
   window.onClose = {counter removeDependant: adapter};
   window.front
};
)
```

Figure 5.39
A dependant that displays the count.

Now one can make displays for any of the counters at any time:

```
~make_display.(~counters[0]);
```

or all of them at once:

```
~counters do: ~make_display.(_);
```

The Observer pattern is considered so important that it is implemented in the class Object and is thus available to all objects, notably in the class variable `dependants` and the methods `addDependant` and `removeDependant`, and also in the example class `SimpleController`. This concept is developed further in the library by the author *sc-hacks-redux*, which is provided in the code examples in this chapter.

5.8 Conclusion

This chapter has attempted to describe the programming language of SuperCollider and its capabilities in as much detail as possible in the given space. It also has presented some techniques of programming that may serve as an introduction to advanced programming. Many other techniques exist. A great many of these are described in print and on the web in publications that deal with design patterns for programming. Kent Beck's *Smalltalk Best Practice Patterns* (Beck, 1996) is recommended as a basic manual of good style, and because the patterns it describes are as powerful as they are small. Gamma et al. (1994) is considered a standard book on patterns. Beck (2000) and Fowler et al. (1999) deal with more advanced techniques of coding.

SuperCollider as an open-source project depends on the active participation of members of the community to continue developing it as one of the most advanced environments for sound synthesis. Contributions by musicians and programmers, through suggestions and bug reports to the scsynth.com forum, through Quarks in the quark repository, or through proposals for inclusion in the SCClassLibrary itself, are vital for the further development of this environment.

One of the most attractive aspects of this environment is that it is equally a tool for music making, experimentation, research, and learning about programming and sound. The features and capabilities of the SuperCollider programming language outlined in this chapter can serve as a springboard for projects that will further expand its capabilities and user base. Whatever the future may bring, the particular marriage of toolmaking and music making that SuperCollider embodies so successfully marks it as an exceptional achievement, and it hopefully will give birth to further original ideas and amazing sounds.

Notes

1. Two relevant definitions of statements are "An elementary instruction in a programming language" (http://www.thefreedictionary.com/statement) and "A statement is a block of code that does something. An assignment statement assigns a value to a variable. A for statement performs a loop. In C, C++, and C# Statements can be grouped together as one statement using curly brackets" (http://cplus.about.com/od/glossar1/g/statementdefn.htm). In SuperCollider, statements enclosed in curly brackets create a function object, which is different from a statement group in C or C++.

2. When a program can construct functions while it is running and store them as objects in variables, it is said that it treats functions as “first class objects” (Burstall, 2000).

References

Beck, K. 1996. *Smalltalk Best Practice Patterns*. Upper Saddle River, NJ: Prentice Hall.

Beck, K. 2000. *eXtreme Programming eXplained: Embrace Change*. Reading, MA: Addison-Wesley.

Burstall, R. 2000. “Christopher Strachey—Understanding Programming Languages.” *Higher-Order and Symbolic Computation,* 13(1/2): 52.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Object Technology Series. Reading, MA: Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Boston: Addison-Wesley.

Strachey, C. 2000. “Fundamental Concepts in Programming Languages.” *Higher-Order and Symbolic Computation,* 13(1/2): 11–49.

6 Events and Patterns

Ron Kuivila

6.1 SynthDefs, Events, and Patterns

SuperCollider's `Event` and `Pattern` classes work together to provide both a concise score language and a flexible set of programming tools. This introduction emphasizes their use as a score language for nonprogrammers. As with other programming approaches in SC, the `SynthDef` class defines the actual sound sources and processors that run on the *synthesis server*. The discussion of SynthDefs is limited to those issues directly relevant to Events and Patterns. It is helpful, but not mandatory, to have reviewed the structure of the synthesis server and to have a basic understanding of Functions, arguments, and variables in reading this chapter. For this reason, it may be helpful to review the earlier chapters of this book before proceeding. Two appendices to this chapter provide a more detailed technical treatment that requires greater knowledge of the language. Chapter 5, with its in-depth overview, may be useful for filling in any gaps.

6.1.1 SynthDefs

A SynthDef assigns a name to a "patch diagram" of unit generators (UGens) and additional names to the patch's control inputs. Figure 6.1 creates a SynthDef named "sine." The synthesis function's arguments (`gate`, `out`, `freq`, `amp`, `pan`, `ar`, and `dr`) define the patch's *control inputs*. The patch is defined in the body of the function using the `UGen` classes that provide language-side representations of the unit generators available on the server.

```
SynthDef(
        "sine",   // name of SynthDef {    // function begins with
 a brace arg gate = 1, out = 0,    // arguments serve as Control
 declarations freq = 400, amp = 0.4,
        pan = 0, ar = 1, dr = 1;
        var audio; audio = SinOsc.ar(freq, 0, amp);    // start wit
h a SinOsc audio = audio * Linen.kr(gate, ar, 1, dr, 2);       // a
pply an envelope audio = Pan2.ar(audio, pan);    // stereo pan, Off
```

```
    setOut.ar(out,audio);     // to bus out and out+1 }
    ).add;     // make and store a SynthDesc and SynthDef
```

Figure 6.1
Example of a SynthDef.

The patch diagram specified by a SynthDef must be downloaded to the server to be used. In addition, a description of a SynthDef's control inputs must be maintained within the language so that Events can play it correctly. The `store` and `add` methods perform this bookkeeping (creating a corresponding `SynthDesc` object for language-side *Synth Description*). The former writes the SynthDef to disk, and the latter does not. (They have equivalents in the `load` and `send` methods, which don't create SynthDescs, though the standard SynthDef method has been `add` since SuperCollider 3.4.) In addition, SynthDefs can be organized into separate libraries rather than simply using the global one. These refinements are discussed in the Help files for the classes `SynthDef` and `SynthDesc`.

6.1.2 Events and Key/Value Arrays

In SuperCollider, parameters are often specified as *key/value* pairs. The key is a Symbol,[1] and the value is any object of the language (usually a Number, an Array, a Symbol, or a Function). For example, when a synth is created, the values of its controls are specified as a *key/value array* consisting of keys interleaved with values (e.g., `[controlName1, value1, controlName2, value2…]`). An `Event` is a collection of key/value pairs that defines a specific action to be taken in response to the message play. By default, Events specify notes to be played on the server.

An Array can be written with a pair of square brackets enclosing its comma-separated elements; an Event is enclosed with parentheses.[2] Within an Event definition, key/value pairs are written by appending a colon as a suffix to the key; Arrays accept this syntax as well. Figure 6.2 creates a key/value array and an event, assigning them to the interpreter variables `a` and `e`, respectively.

```
a = [ type:   \note, instrument:  'sine', freq:     400, amp:
 0.1,
   pan:      0, ar:       2, dr:       4, sustain: 2
];
e = ( type:    \note, instrument:  'sine', freq:     400, amp:
0.1, pan:      0, ar:       2, dr:       4, sustain: 2
);
e.play; // play the note
```

Figure 6.2
Example of a key/value `Array` and a note `Event`.

When a note event is sent a `play` message, it looks up the description (the SynthDesc) of the SynthDef identified by the value of the key `\instrument`. It uses that description to determine the names of the SynthDef's controls and uses those *control keys* to generate the Open Sound Control (OSC) commands needed to create and release a synth that plays the SynthDef. It then schedules those commands to be sent to the server at the appropriate times. An array of those OSC commands and their time offsets can be created by sending the event an `asOSC` message:

```
e.asOSC.do {| osc | osc.postcs};
```

This would lead to the result posted here:

```
[0.0, ['s_new', 'sine', 1000, 0, 1, 'out', 0, 'freq', 400.0,   'am
p', 0.1, 'pan', 0, 'ar', 2, 'dr', 4]]
[2.0, ['n_set', 1000, 'gate', 0]]
```

Events also specify a time increment that is returned in response to the message `delta`. The increment returned is normally determined by taking the product of the values of the keys `\dur` and `\stretch`. This timing value is used to determine the rhythm of events specified by *Event patterns*.[3]

6.1.3 Sequences of Values and Events

`Patterns` specify sequences of values. For example, the class `Pseq(array, repeats, offset)` is used to define a pattern that specifies a sequence created by repeatedly iterating over the entirety of the array beginning at offset. More concretely, the pattern `Pseq([0,1,2,3,4,5,6,7], 3, 1)` specifies the sequence `[1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0]`.

Figure 6.3 illustrates how `Patterns` can be combined arithmetically and can be used within the definition of other patterns. In these examples, the `asStream` message is sent to the pattern to obtain a `Stream`. The sequence of values that the stream generates is collected into an array with the `nextN` message.

```
   ~pat = Pseq((0..2), 3, 1); // the pattern ~stream = ~pat.asStrea
m;    // the stream ~stream.nextN(10);            // obtain 1 extra
value
// ~stream returns: [1, 2, 0, 1, 2, 0, 1, 2, 0, nil]
// create a new pattern as an arithmetic combination of
// the original pattern with itself ~reusedPat = ~pat * 10 + ~pat;
```

```
 ~stream = ~reusedPat.asStream; ~stream.nextN(10);
// stream returns: [11, 22, 0, 11, 22, 0, 11, 22, 0, nil]
// Use ~pat1 in the defin ~pat2 ~pat1 = Pseq((0..2)); ~pat2 = Pseq
([~pat1, 2 * ~pat1, 3 * ~pat1]); ~stream = ~pat2.asStream;
   ~stream.nextN(10);   // 1 more value than the sequence specifies
// stream returns: [0, 1, 2, 0, 2, 4, 0, 3, 6, nil]
```

Figure 6.3
Defining a `Pattern`, creating a Stream and extracting its values.

Patterns that specify sequences of events are called *Event patterns,* and they can specify complete musical sequences. The class `Pbind(key, pattern, key2, pattern2…)` defines Event patterns by *binding* different *value patterns* to different event keys. Each Event in the sequence specified begins as a copy of a *prototype event*. Then the next value of each value pattern[4] is taken and assigned to its corresponding key to produce the next element in the `Pbind`'s sequence. That sequence ends when any of its constituent value patterns end.

When defining a `Pbind` object, the key/pattern pairs can be written as a key/value array[5] or as symbols interleaved with patterns; both are illustrated in figure 6.4, which defines a pattern that will produce the first 8 overtones of 100 Hz in an arpeggio and then stop.

```
// 1.
Pbind (*[ dur:      0.2, freq:    Pseq([100, 200, 300, 400, 500, 60
0, 700, 800])
]);
// 2.
Pbind( \dur, 0.2, \freq,   Pseq([100, 200, 300, 400, 500, 600, 700,
800])
);
```

Figure 6.4
Two ways of writing the same Event pattern.

In figure 6.5, an event pattern is assigned to the Interpreter variable *p*; it specifies a repeated sequence of the first 11 notes of the overtone series with a 4-event accent pattern, a 7-event duration pattern, and a 5-event sustain pattern. The pattern is then rendered as a sound file with a render message[6] and played in real time with a `play` message.

```
// render can only find SynthDefs in SynthDef.synthDefDir
// So synthdefs must be stored rather than added to be put in // th
```

```
at dir.
SynthDescLib.default[\default].def.store;
p = Pbind(*[ instrument:   \default, detune:   [0,1,3], freq: Pseq
 ((1..11) * 100, 4 * 5 * 7), db: Pseq([−20, −40, −30, −40], inf), p
an: Pseq([−1,0,1,0], inf), dur:      Pseq([0.2,0.2,0.2,0.2, 0.4,0.
4,0.8], inf), legato:  Pseq([2,0.5,0.75,0.5,0.25], inf) ]);
// render 40 seconds of the pattern in the file named "sf.aif" p.re
nder("sounds/sf.aif", 40)
// now play the pattern in real-time p.play;
```

Figure 6.5
A more elaborate Event pattern.

We detail next how patterns can be defined as combinations of other patterns. Programmers familiar with Routines can use the class `Pspawner(routineFunction)` to dynamically create sequential and parallel combinations of event patterns.

6.2 The Default Event

Event and its parent classes Environment and IdentityDictionary have a particularly wide range of application within SuperCollider (detailed in chapters 5 and 20). Here, we focus on the structure of Event's default mechanism as it is used in conjunction with the patterns library. This default mechanism is implemented as a prototype Event with an extensive collection of predefined keys.

6.2.1 Event Types

The default Event contains an extensible collection of *event types* that specify different actions to be taken in response to play. Using an event involves setting the correct type and overriding the value of any keys whose default values are inappropriate. By default, the Event type is `\note`, which plays a note. The remainder of this section may be skipped on first reading, as our primary focus will be on the note event type discussed in section 6.2.2.

Table 6.1 offers a reference list of default event types and their corresponding OSC commands.[7] Most event types provide a direct interface to an OSC command and can be understood by consulting the Server-Command-Reference Help file.

Table 6.1
`Event` types of the default event

Type	OSC	Relevant Keys	Action
note	s_new	see below	Create a synth with release.
on	s_new	server, group, addAction, id	Create a synth without release.

Type	OSC	Relevant Keys	Action
set	n_set	server, id, args	Set values of controls named by args.
off	n_set	server, id, hasGate	Release a node by setting gate to 0.
	n_free		If there is no gate control, free it.
group	g_new	server, group, addAction, id	Create a group.
kill	n_free	server, id	Free a node (usually a group).
bus	c_setn	server, id, array	Send array to consecutive control buses, starting at id.
alloc	b_alloc	server, bufnum	Allocate buffer.
free	b_free	server, bufnum	Free buffer.
gen	b_gen	server, bufnum, gencmd, genflags, genarray	Generate values in buffer.
load	b_allocRead	server, bufnum, frame, filename, numframes	Allocate and load file to buffer.
read	B_read	server, bufnum, frame, filename, numframes, bufpos, leaveOpen	Read file into preallocated buffer.
rest	—	delta	Defines a rest.

Table 6.2 lists the keys that determine event timing; table 6.3 lists the keys that control the transmission of OSC commands;[8] and table 6.4 lists the keys that specify the creation of groups and synths on the server.

Table 6.2
`Event` timing keys

`Dur`	Time until next note, defaults to 1.0
`Stretch`	Expansion/contraction of durations, defaults to 1.0
`Tempo`	Tempo of the `TempoClock`, defaults to `nil`, leaving current tempo
`timingOffset`	A delay imposed before the event is played, in beats
`Lag`	A delay imposed before the event is played, in seconds
`Delta`	A delay until the next event in the sequence, usually `~dur * ~stretch`

Table 6.3
Command transmission keys

`Server`	Defaults to `nil`, in which case `Server.default` is used
`schedBundle`	Sends bundle to server or accumulates into a score
`schedBundleArray`	Same as `schedBundle`, but with the commands in an array
`schedStrummedNote`	Implements strumming, using `schedBundle`

Table 6.4
Node-related keys

`Id`	nodeID of the synth or group, often left nil for dynamic allocation.
`Group`	The nodeID of the synth's placement target (group or synth) default value is 1; the nodeID of the "default group."

`Id`	nodeID of the synth or group, often left nil for dynamic allocation.

`addAction`	The placement of the synth in the node tree on the server, defaults to 0. It can be specified using integers or symbols, as listed elsewhere, and defined by the class Node.
	`\addToHead \h 0`
	`\addToTail \t 1`
	`\addBefore 2`
	`\addAfter 3`
	`\addReplace 4`

`Instrument`	The name of the SynthDef to be played, defaults to `\default`
`Out`	Output bus for the synth, defaults to 0

In figure 6.6, a group event creates a group with nodeID 2, a note event plays a note within that group, then all the notes in the group are released and the group itself is freed.

```
( (type:   \group,
id: 2
).play;  // create a group with nodeID 2
   (    type:    \note,    // play note sustain: 100,      // lasti
ng 100 seconds
        group:    2         // in group 2 ).play;
)
( (type:   \off,   id: 2).play;      // release all notes in the gr
oup (type:   \kill,   id: 2, lag: 3).play;   // and free the group
 3 seconds later
)
```

Figure 6.6
Using Event types

6.2.2 Keys for Note Events

The note event types (`\note`, `\on`, `\set`, `\off`, `\kill`) play and release synths on the server. By convention, SynthDefs reserve the control names `amp`, `pan`, `sustain`, and `freq` to set the volume, position (usually in a stereo field), duration, and pitch of the note to be played, respectively. For each of these control keys, the default event provides a more abstract interface. For example, note amplitude can be set by assigning a multiplier between 0 and 1 to the key `\amp` or a decibel value between −100 and 0 to the key `\db`. Similarly, note sustain can be set directly through `\sustain`, or implicitly as the product of `\dur` and `\legato`.

Tables 6.5 through 6.7 enumerate keys that specify the amplitude, articulation, and pitch of notes. These keys create additional layers of abstraction that allow note properties to be specified in different ways.

Table 6.5
Amplitude-related keys

`Amp`	Amplitude as a multiplier
`Db`	Amplitude expressed in decibels (0 dB is the maximum level)
`Pan`	Pan position:–1 is left, 0 is center, 1 is right

Table 6.6
Note articulation-related keys

`Sustain`	Time until note is released
`Legato`	Ratio that determines sustain in relation to `dur * stretch`
`Dur`	Time until next note in the pattern
`Gate`	If it is a control of `instrument`, it is set to 0 to release the note
`Trig`	Used to trigger and release an envelope without releasing the note

Table 6.7
Frequency-related keys

`stepsPerOctave`	Defines the equal tempered gamut (normally 12)
`scale`	Array of intervals within the equal tempered gamut
`root`	Fractional transposition within the equal tempered gamut
`degree`	Pitch as a scale degree within the scale `scale`
`mtranspose`	Modal transposition of degree within a scale
`note`	Position within the equal tempered gamut
`gtranspose`	Fractional transposition within the equal tempered gamut
`octave`	Default transposition of note within twelve-tone chromatic scale
`midinote`	Fractional MIDI key# (69 -> 440, 69.5 is a quarter tone sharp)
`ctranspose`	Fractional transposition within the 12-tone scale
`freq`	Pitch directly as a frequency (in hertz)
`harmonic`	Is multiplied by the frequency determined by midinote
`detune`	Usually 0 or an array of values to “fatten” the sound
`detunedFreq`	The function that determines the value of freq used to create the note; by default, `{~freq * ~harmonic + ~detune}`

These layers of specification are implemented with functions. Each function computes its value in terms of the next higher level of representation. For example, the value of the key `\amp` is a function that converts the value of the key `\db` into an amplitude multiplier. Assigning a value directly to `\amp` eliminates the function, overriding the `\db` specification. This is true in general: assigning a value to a key overrides any related higher-level specifications.

The pitch specification is elaborately layered, and pitch can be determined as follows:

- A `degree` within a `scale`

- A `note` in a gamut of `stepsPerOctave` equally tempered steps
- A `midinote` in a twelve-tone pitch set, or
- Directly as a frequency, using the key `freq`.

Each layer has an associated transposition value:

- `root` transposes `scale`.
- `mtranspose` transposes `degree` within `scale`.
- `gtranspose` and `octave` transpose `note` within the gamut defined by `stepsPerOctave`.
- `ctranspose` transposes `midinote` within equal temperament.
- `harmonic` transposes `freq`.

Assigning a value to a pitch key overrides both the higher-level pitch keys and their associated transpositions. Assigning a Symbol to a pitch key that has not been overridden specifies a rest.

So long as the SynthDef has a properly stored `SynthDesc`, note events will automatically send a synth all its controls, whatever their names. Nevertheless, it is best to reserve default keys for their predefined purposes. There is little utility in using the keys `freq`, `amp`, `pan`, `out`, `in`, and `trig` for anything other than their default purposes; changing the significance of `server`, `group`, `addAction`, or `instrument` will break the default mechanism altogether.

6.2.3 Note Events and Chords

Assigning an array to a control key (i.e., a key that actually names a control in the SynthDef identified by `instrument`) produces a chord. If several keys are assigned arrays, the event plays as many notes as the largest array. For each note in the chord, key values are obtained by iterating through whatever values were assigned. This is implemented by sending the key/value array generated by the event a `flop` message.[9] For example, here is a key/value array with arrays as values:

```
[
  freq: [100, 200, 300, 400], amp: [0.1, 0.2], pan: [-1, 0, 1]
]
```

After receiving a flop message, it becomes

```
[
  [freq: 100, amp: 0.1, pan: -1],
  [freq: 200, amp: 0.2, pan: 0],
  [freq: 300, amp: 0.1, pan: 1],
```

```
    [freq: 400, amp: 0.2, pan: -1]
]
```

Figure 6.7 provides examples of Events using this feature.

```
// 2nd inversion—e loudest
(degree: [—3,0,2],  sustain: 2, db: [—20, —20, —10]).play
// 2nd inversion—c loudest
(degree: [—3,0,2],  sustain: 2, db: [—20, —10, —20]).play
// note "fattened" by three detuned copies
(degree: 0,          sustain: 2, detune: [0,3, 5]).play
// each detune is assigned to a different pitch, fat free.
(degree: [—3,2,4],  sustain: 2, detune: [0,3, 5]).play
// detune rotates through each note in the chord
(degree: [—3,2,4],  sustain: 2, detune: [0,0,0,3,3,3,5,5,5])   .pla
y
```

Figure 6.7
Chord Events.

6.3 Patterns

6.3.1 An Overview of Patterns and Streams

Patterns are analogous to musical notation; they are abstract representations of sequences independent of any specific performance. Any object in the language can be viewed as a pattern, but most patterns specify a sequence whose values are simply the object itself. We will refer to these as *trivial patterns,* in contrast to *nontrivial patterns* that specify sequences with changing values. Numbers and Symbols are examples of trivial patterns, whereas classes such as `Pseq` are used to define nontrivial patterns.

To play a pattern, it is sent an `asStream` message. This returns a `Stream` object that will generate the sequence the pattern specifies. It does so 1 element at a time in response to the message next. Streams created by event patterns are called *event streams*. When patterns are treated as a score language, these details can be largely ignored. (We will return to them later, in the context of real-time performance.)

Tables 6.8, 6.9, and 6.10 list representative pattern classes, grouped according to the values of the sequences that instances of those classes will specify. An initial source of confusion is that many Pattern classes create patterns of patterns. For example, `Pseq` sequences produce whatever is in their array, which could be numbers, event patterns, or any other objects in the language. The important point is that such *pattern patterns*

can be used to specify sequences of values assigned to a key within a `Pbind` or sequences of entire event patterns defined with `Pbind`.

Table 6.8
Numerical pattern classes

`Pwhite(lo, hi, repeats)`	Uniform random values between lo and hi
`Pbrown(lo, hi, step, repeats)`	"Brownian motion"; step is the maximum jump
`Pseg(array, dur, curves, repeats)`	Break-point envelope; array must be numerical

Table 6.9
`Event` pattern classes

`Pbind(key, pat, key2, pat2…)`	Bind patterns to keys
`Ppar(eventPatternArray, repeats)`	Play the event patterns in list in parallel
`Pchain(pat1, pat2,…)`	Compose multiple patterns
`Pmono(name, key, pat…)`	Start a single synth and control it

Table 6.10
`Pattern` pattern classes

`Pseq(array, repeats, offset)`	Iterate entire array repeat times, starting at offset.
`Prand(array, repeats)`	Randomly choose from array repeat times.
`Pstutter(pattern, repeat)`	Repeat each value of the pattern repeat times.
`Pstep(array, dur, repeats)`	Iterate the array, holding each value for dur seconds.
`Pfunc(function, resetFunction)`	The entire function defines sequence elements.
`Prout(function)`	Uses `embedInStream` to embed individual values or entire subsequences into the stream.

6.3.2 Combining Patterns

Patterns can be used recursively in the definition of other patterns. For example, the arguments `lo` and `hi` of the random-value pattern `Pwhite` could be patterns defined with `Pseq` or `Pstep` or `Pseg`.

In addition, numerical patterns can be arithmetically combined to define new patterns. Combinations can be made using any of the unary and binary messages defined by `AbstractFunction`:

```
a = Pseq([1, 2, 3], 1); // iterate 1, 2, 3 once
b = Pseq([a, 3, 2, 1], 2);   // pattern defined with another patter
n
a + b; // sum of patterns
a * b * 33;     // product of patterns
```

```
midiratio(b);     // modified by unary message midiratio
a round: 4        // modified by binary message round
```

Event patterns cannot be combined arithmetically. Instead, sequential and parallel combinations are made using the patterns `Pseq` and `Ppar`.

```
a = Pbind(*[dur: Pseq([0.4], 5)]);
b = Pbind(*[degree: Pseq([10, 6], inf), dur: Pseq([0.5], 4)]);
Pseq([a, b], 2).play;
Ppar([a, b], 4).play;
```

6.3.3 Time-Based Pattern Classes

Most patterns define sequences of values that are independent of time; the patterns `Pstep` and `Pseg` specify sequences that are sampled as functions of time. They are useful for attributes, such as dynamics and chord progressions, that are most easily described independent of the specific rhythmic sequences that articulate them. In conjunction with the patterns `Ppar` and `Pchain` (discussed below) they can be used to specify attributes shared by different patterns running concurrently.

6.3.4 Interdependent Key Values in Event Patterns

`Pbind` allows completely independent patterns to specify key values, but it is often more natural to use a single sequence that determines several key values at once. For example, a melodic phrase may be better represented as a sequence of note/duration pairs than as 2 independent sequences. To do this, a pattern can be bound to an array of keys rather than a single key. In this case, the pattern specifies a sequence of arrays. The elements of each value array are assigned to the corresponding keys in the key array:

```
Pbind(*[
  #[degree, dur],  Pseq([[0, 1], [3, 1/2], [6, 1/3], [8, 1/4],
 [7, 1]]),
  db:  -20
]) .play
```

When a `Pbind` is played, each `pattern` bound to a key (more precisely, the stream generated by that `pattern`) is advanced in turn. This makes it possible for a value pattern to determine the values of any keys bound earlier in the key/pattern array. The pattern `Pkey(key)` is a pattern that reads those values.[10] Figure 6.8 shows an event pattern whose pitches are the first 16 overtones of 100 Hz. The duration of each note is directly proportional to its harmonic number, and its volume is inversely proportional.

The whole texture is stretched with a `Pseg` scaled to range from 1 to 0.125, effectively octupling the tempo.

```
    Pbind(*[ stretch:   Pseg([0,0.1,0.2,1],8).linexp(0,1, 1,0.125),
 midinote:  100.cpsmidi, harmonic:  Pwhite(1, 16), legato:    Pkey
(\stretch) * Pkey(\harmonic)/2, db:          −10−Pkey(\harmonic), de
tune:    Pwhite(0.0,3.0), dur:        0.2, ]).play
```

Figure 6.8
Interdependent key values in a Pattern.

The pattern `Pchain(pat1, pat2, ..)` allows the events of one pattern to be defined as a combination of values generated by others. In this context, the time-based patterns can be used to define attributes shared by patterns running in parallel. In figure 6.9, pattern `a` specifies a changing scale, and patterns `b` and `c` define notes and chords within a scale. Pattern `d` chains `a` to the parallel combination of `b` and `c`, so both share the same scale.

```
a = Pbind(*[ scale:   Pn (Pstep([[0,2,4,5,7,9,11], [0,1,3,5,6,8,1
1]], 5)), db:       Pn(Pseg([−20, −30, −25, −30], 0.4))
]);
b = Pbind(*[ degree: Pbrown(0, 6, 1), mtranspose: Prand([\rest, Pse
q([0], 5.rand)],inf), dur: 0.2, octave: 6
]);
c = Pbind(*[ degree: [0,2,4], mtranspose: Pbrown(0, 6, 1), dur: 0.
4, db: −35
]);
d = Pchain(Ppar([b, c]),a);
d.play;
```

Figure 6.9
Chaining `Event` patterns.

Events in a Pchain are computed from right to left. The event returned by the rightmost pattern is used as the protoype `event` for the pattern to its left, the result of that pattern is applied to the pattern to its left, and so forth. Patterns to the left can override any values received from patterns to the right. There is one exception: `Ppar` calculates the `delta` of its subpatterns in order to schedule them. Thus it is not possible to alter the `\dur` key of an event coming from a `Ppar`. Any such changes must be done by patterns to the right of the `Ppar` in the `Pchain`.

6.3.5 Defining Patterns with Pfunc and Prout

Pfunc and Prout make it possible to define patterns with a function. Within a Pfunc, the function receives the event as an argument and returns a value. For example, here is a Pfunc that duplicates the functionality of Pkey: Pfunc({|ev| ev[\aKey]}).

Within a Prout, the function returns values by sending them an embedInStream message. Trivial patterns respond to embedInStream by embedding themselves as a single-valued sequence; nontrivial patterns embed the entire sequence they specify. Thus, a Prout can select or even define new patterns on the fly and embed them in the current sequence.

In the following example, the function defines its own events and embeds them in the event stream. The first event in the stream appears as the argument of the function, and subsequent events appear as the return value embedInStream. The function stops at each embedInStream message and resumes immediately after, so the sequence of expressions in the function directly determines the sequence of events in the pattern:

```
Prout({|ev| // modifies protoEvents
  ev = (freq: 400).embedInStream(ev.copy);
  ev = (freq: 500).embedInStream(ev.copy);
  ev = (freq: 600).embedInStream(ev.copy);
  ev = (freq: 700).embedInStream(ev.copy);
  ev = (freq: 800).embedInStream(ev.copy);
}).play;
```

Figure 6.10 shows a more elaborate example, in which the Prout creates a reference pattern that it alternates with new patterns created on the fly in the midst of performance.

```
Prout({| ev |
   var pat, refPat; refPat = Pbind(*[dur: 0.2, note: Pseq([0,0, 0,
 7,0, 7])]);
   loop {
        ev = refPat.embedInStream(ev);
   pat = Pbind(*[ dur: [0.2, 0.4].choose, note: Pseq(Array.fill(5,
 {10.rand}), 3.rand) ]); ev = pat.embedInStream(ev);
   }
}).play
```

Figure 6.10
Using Prout to define and play patterns on the fly.

In figure 6.11, ~patA creates an array of notes and alters a randomly selected note in the array each time the array has been played. In ~patB, an event pattern is selected at

random to be embedded. These patterns are combined with `Ptpar`, which schedules the entrance of two copies of `~patA` and one copy of `~patB`. This parallel combination is faded in and out, using `Pchain`.

```
~patA = Pbind(*[ dur:   0.2, degree: Prout({| ev | var noteArray =
(0..5); loop { ev = Pseq(noteArray) .embedInStream(ev); noteArray
[6.rand] = 7.rand; } })
]);
~patB = Prout({| ev | var pat, pats= [ Pbind(* [degree: Pseq([0,
 7]), dur: 0.2]), Pbind(* [degree: Pseq([11, 7]), dur: 0.2]), Pbind
(* [degree: Pseq([16, 7]), dur: 0.2]). (type: \rest, delta: 1)
   ];
   loop { pat = pats.choose; ev = pat.embedInStream(ev); }
});
Pchain( Pbind(*[
        db:  Pn(Pstep([−15, −25,−25, −20, −30, −25], 0.2))
        + Pseg([−30, −5,−10, −40], 12) ]), Ptpar([ 0, ~patA, 0, ~pa
tA, 12, ~patB ])
).play;
```

Figure 6.11
Using `Prout` to define value and event patterns.

6.3.6 Rendering Event Patterns

An event pattern can be rendered into a sound file either by sending it a `render` message directly or by creating a `Score` object with the `asScore` message and rendering that score with render. The latter approach allows an alternative `protoEvent` to be used and for the score to be offset in time for subsequent manipulation. Here are those methods and their arguments:

```
render(path, maxTime, sampleRate, headerFormat,
        sampleFormat, options, inputFilePath)
   path is the path name for the resultant audio file.
   headerFormat is "AIFF" by default; could be "WAV" or "SD2"
   sampleFormat is "int16" by default, could be "int24",   "int32",
or "float"
   asScore(duration, timeOffset, protoEvent)
   duration and timeOffset are self-explanatory.
   protoEvent defines the prototype event used by the pattern
```

Figure 6.12 defines a pattern, turns it into a score, renders the score, and then plays that file.

```
~pattern = Pbind(*[ instrument:   "default", freq:    Pseq([100, 20
0, 300, 400, 500, 600, 700, 800, 900,   1000, 1100], 5), db:
 Pseq([-10, -30, -20, -30], inf), dur:      Pseq([0.2,0.2,0.2,0.2,
 0.4,0.4,0.8],inf), legato: Pseq([2,0.5,0.75,0.5,0.25], inf)
]);
~score = ~pattern.asScore(24 * 11/7);
~score.render("recordings/test.aif");
SoundFile("recordings/test.aif").play;
```

Figure 6.12
Rendering and playing a pattern.

6.4 Timing Considerations

6.4.1 Timing and Articulation

Music is filled with examples of events and actions that, although conceived as simultaneous, actually occur at different times. Notes may be played before or after a beat, chords can be broken or strummed, signals can be given a beat ahead, and so forth. The `\lag` and `\timingOffset` keys set delay times between the time an event receives its `play` message and the time the resultant OSC commands are sent to the server. Lag is expressed in seconds; `timingOffset` is in time units altered by the value of `tempo`.

For file rendering, notes that occur off the beat can be specified simply by assigning a positive or negative value to `\lag` or `\timingOffset`, and the key `\strum` sets a delay time between notes in a chord:

```
(lag: 0, strum: 0.1, note: [12, 16, 19], sustain: 1).play;
```

A negative strum value will cause the notes in the chord to precede the beat. If the key `\strumEndsTogether` is set to `true`, individual notes will be lengthened or shortened to end the chord at the time specified by `\sustain`.

Real-time performance requires special measures, described in section 6.5.2.

6.4.2 Audio Rate, Control Rate, and Sample Accurate Scheduling

As noted in chapter 2, SuperCollider, like most software synthesis systems, computes audio in blocks of samples. This introduces a distinction between *audio rate*, the sample rate of audio samples, and *control rate*, the rate at which sample blocks are computed. A 44.1-kHz sampling rate with a block size of 64 will yield approximately a

690-Hz control rate and a 0.00145-s control period. The server responds to OSC commands at the control rate.

Starting synths at sample block boundaries can create objectionable artifacts in the sound. The UGen `OffsetOut` provides sample accurate scheduling by using the time stamp of the synth's creation command to determine how many samples to provide to its first sample block. This places the beginning of the synth's audio at the correct time.

A related issue is that synths can respond to control changes only at the control rate. Attempts to gate an envelope with a duration less than the control period will have undefined results (i.e., the note may not occur or may sustain indefinitely). With very tightly articulated sounds, it is best to use fixed-duration envelopes running at the audio rate that are time-scaled by the sustain value of the event. In this case, the event type `\on` can be used with the key `\id` set to `−1` (which indicates to the server that the language will not attempt to communicate with the synth). Granular synthesis is an example of an approach in which these timing issues are extremely important. Figure 6.13 granulates a sound file, steadily increasing grain size from 0.01 s to 2 s, with an overlap of four grains.

```
SynthDef("playbuf", {| out=0, bufnum = 0, rate = 1,
   startPos = 0, amp = 0.1, sustain = 1,
   pan = 0, gate = 1|
   var audio, env;
   rate = rate * BufRateScale.kr(bufnum);
   startPos = startPos * BufFrames.kr(bufnum);
   env = EnvGen.ar(Env.sine, 1, timeScale: sustain, doneAction: 2);
   audio = PlayBuf.ar(1, bufnum, rate, 1, startPos, 0);
   audio = env * audio;
   audio = Pan2.ar(audio, pan, amp);
   OffsetOut.ar(out, audio);
}).add;

Pseq([
   (    type:      \load,
        filename:  "sounds/a11wlk01.wav",
        bufnum:    1,
        delta:     0
   ),

   Pbind(*[
        instrument:   "playbuf",
        type:        \on,
        id:          −1,
```

```
        dur:         Pseg([0,1],21).linexp(0,1,0.01,2),
        legato:     4,
        startPos: Pn(Pseg([0,1], 10)),
        bufnum:     1,
  ]),
  (     type:     \free,
        bufnum:     1
   )
]).play(quant: 0)
// the result: [('midinote': 60), ('midinote': 64), ('midinote': 6
7)]
```

Figure 6.13
Sound-file granulation with a `Pattern`.

Chapter 16 provides a detailed exploration of granular synthesis and related techniques in SC.

6.4.3 Freeing Synths

The single most common mistake made when first working with patterns is to write a SynthDef that does not free created synths when they are done. Although not strictly a timing issue, the main impact of this mistake is to overload the server and destroy the integrity of the audio output. The following example illustrates the problem and its solution; the Help file UGen-doneActions provides further details:

```
SynthDef("eternal", {|out, freq, amp, pan, gate|
   var audio, env;
   audio = SinOsc.ar(freq, 0, amp);
   env = Linen.kr(gate);   // mistake: does not delete synth
// env = Linen.kr(gate, doneAction: 2);   // doneAction // of 2 is
 needed
   audio = audio * env;
   audio = Pan2.ar(audio, pan);
   Out.ar(out,audio);
});
```

6.5 Real-Time Performance and Interactive Control of Patterns

6.5.1 TempoClocks and Quantization

Here is `Pattern's` play method with its associated arguments:

```
play(clock, protoEvent, quant)
   clock is a TempoClock, and its default value is TempoClock.defau
lt protoEvent is an Event used to set default values for the patter
n quant is a quantization value that constrains when the pattern wi
ll begin playing
```

The method returns an `EventStreamPlayer` object, which provides interactive control through the messages `play`, `pause`, `resume`, and `stop`. The `EventStreamPlayer` creates a stream out of the pattern and then plays it, using the specified clock to determine event timing and the specified protoEvent (or the default `Event`) as the starting point for each event that the stream produces.

As noted in chapter 3, a `TempoClock` has a controllable `tempo`, so its logical time is expressed in *beats* rather than seconds. `Event` patterns can set the clock tempo through the key `\tempo`. Any other patterns that share the same clock will experience the same tempo change. In contrast, the `\stretch` key affects only the pattern that it is defined within.

The specific time at which the pattern begins playback depends on the value of `quant`. If `quant` is 0, performance begins as soon as the message is sent. Otherwise, the pattern begins at the beat that is the earliest integer multiple of quant.

Sometimes simple quantization is not enough. The `quant` argument can also be a two value array that specifies `quant` and `phase`. `phase` is an offset relative to `quant`. For example, if `quant` is 1 and `phase` is 0.75, the pattern will commence playing 1/4 beat before the basic beat time.

A pattern may need to play at a `timingOffset` as well. This sets a delay time between the computation of an event and the transmission of the OSC commands that it generated, providing some leeway for notes that play ahead of the beat. Such notes can be scheduled by setting the `\timingOffset` key to the sum of its current value and the desired (i.e., negative) delay. So long as this sum remains nonnegative, correct timing will be maintained.

As a three-value array, the `quant` argument specifies `quant`, `phase`, and `offset`. This provides the specified `timingOffset` to the entire `EventStream` while guaranteeing that its first note sounds at the time specified by `quant` and `phase`. As a convenience, it is also possible to specify the `timingOffset` in the prototype event used by the `EventStreamPlayer`.

6.5.2 Compensating for Delays between the Language and the Server

In SuperCollider, the language is decoupled from synthesis in order to prevent timing problems from creating “glitches” in audio output. This creates a need for additional

layers of timing specification that enable the language to specify different trade-offs between timing accuracy and fast execution.

In the real-time context, synchronization between the language and the server is maintained by a network time base. Commands sent by the language have time stamps that indicate when they should be performed. Because network connections can have delays that vary from message to message, a latency is usually added to these time stamps. If the latency is larger than the longest network delay, relative timing will be accurately maintained. This value is kept in the latency instance variable of the `Server` object. By default the latency is 0.2 seconds.

If the Server and language do not share a network time base, time stamps cannot be used and the Server object's `latency` should be set to `nil`. This exposes an additional nuance in the actual timing of audio generation. Audio interfaces are often configured to maintain sample buffers that are much larger than typical sample block sizes. In this case, the server must compute enough sample blocks to fill the buffer. Time stamps enable the server to respond to commands at the correct time in spite of the timing variations that these large buffers create, but those variations will be exposed when using an unsynchronized server. Reducing the hardware buffer size to the block size will correct this.

6.5.3 Conduction with PatternConductor

When an `EventStreamPlayer` is paused or stopped, it may have sustaining notes. These notes can be left at their intended durations, released immediately, or, in the case of a pause, sustained until the `EventStreamPlayer` is resumed. Note releases are scheduled by the `EventStreamPlayer's` clock, so all these possibilities can be realized by adjusting its tempo. `PatternConductor` is a variant of `EventStreamPlayer` that creates its own `TempoClock` and provides a tempo argument for its pause and stop methods.

6.6 Appendix 1: How Patterns Are Performed by Streams

6.6.1 Defining Streams with Patterns and Routines

Sending a pattern an `asStream` message creates a `Stream` object that generates an instance of the sequence that the pattern specifies. The sequence is generated one element at a time in response to a sequence of `next` messages. Trivial patterns simply return themselves in response to both `asStream` and `next`. Thus, a trivial `Pattern` is also a trivial `Stream`.

Here, we define a `Pseq`, create a stream from it, and assemble the stream's values into an array:

```
a = Pseq([1, 2, 3]).asStream;
[a.next, a.next, a.next, a.next];
// [1, 2, 3, nil] // the result
```

Once a stream runs out of values, it returns `nil` in response to `next`. Since the pattern specified a 3-element sequence, the fourth element of the array is `nil`.

The class `Routine` is used by patterns to create their associated streams. As noted in chapter 3, a `Routine` is a `Function` that can return from any point in its definition and, when called again with a next message, resume where it left off. The messages `yield` and `embedInStream` define the exit and entry points, respectively, within the function used to define the routine. For trivial patterns (and streams) `embedInStream` is identical to `yield`; it yields the object once and returns. For a nontrivial pattern, it performs `asStream` and *embeds* the resultant stream in the routine. The stream maintains control until it has yielded all the values in its sequence in response to `next` messages sent to the routine. Figure 6.14 illustrates the difference between `yield` and `embedInStream`, and figure 6.15 presents the definition of the Routine used by `Pseq(array, offset, repeats)` to create its associated streams.

```
r = Routine{
    Pseq([1,2,3]).yield;
    Pseq([1,2,3]).embedInStream;
    123445.embedInStream;
    123445.embedInStream;
};

[next(r), next(r), next(r), next(r), next(r), next(r)];
// the result: [a Pseq, 1, 2, 3, 123445, 123445, nil]
```

Figure 6.14
`Yield` versus `EmbedInStream`.

```
Routine({
    repeats.value.do({
        list.size.do({arg i;
        item = list.wrapAt(i + offsetValue);
        inval = item.embedInStream(inval);
        });
    });
});
```

Figure 6.15
The definition of the stream created by Pseq.

As the stream plays, each element in the array is sent the embedInStream message. Thus each element of the array can be a trivial pattern, a nontrivial pattern, or even a stream. This makes it possible to define patterns within patterns to arbitrarily deep levels of nesting without explicitly invoking any methods. To a large degree, this accounts for the remarkable concision of pattern definitions.

6.6.2 EventStream and ScoreStream Players

An Event pattern is performed by an EventStreamPlayer. The player sends the pattern an asStream message that creates the EventStream that actually generates the pattern's event sequence. The player obtains these events by sending the stream a next(protoEvent) message, which returns the protoEvent as altered by the event stream. It plays the event and then sends the EventStream a delta message to determine the delay until its next event. It uses this time increment to reschedule itself within the time base of its TempoClock.

Scores and rendered files are created with a ScoreStreamPlayer. This object simulates the role of both a Server (for id allocation) and a TempoClock (for tempo control) and alters the \schedBundle and \schedBundleArray keys of the protoEvent to collect OSC commands and their time stamps rather than sending them to a server.

6.7 Appendix 2: Event and Its Superclasses

6.7.1 The Class Derivation of Event

Event is derived from the classes Dictionary, IdentityDictionary, and Environment. Dictionary defines the methods at(key) and put(key, value), which allow the values of existing keys to be examined and altered and new key/value pairs to be added.

IdentityDictionary implements a faster lookup mechanism, in which equivalent keys must be *identical objects*. Because of this, Symbols are generally used as keys. IdentityDictionary also defines the instance variables parent and proto. The method at attempts to retrieve a key's value first from the dictionary, then from its protodictionary, and finally from its parent dictionary, returning the first non-nil value that it finds. This provides the basis for Event's default mechanism.

Environment implements features akin to "name spaces" in other languages. When the language starts, it creates and stores an Environment in topEnvironment and currentEnvironment (class variables of Object, which are globally accessible). The topEnvironment variable is generally left unchanged, so it can provide a globally

accessible name space. In contrast, `currentEnvironment` is often altered to provide access to different local name spaces. The compiler provides a shortcut syntax in which `~` replaces `currentEnvironment`, making the following expressions equivalent:

```
// "getter" messages that retrieve a value from a key within // cur
rentEnvironment
   ~key;
   currentEnvironment[\key];
   currentEnvironment.at(\key);
   // "setter" messages that set the value of a key within // curre
ntEnvironment
   ~key = 888;
   currentEnvironment[\key] = 888;
   currentEnvironment.put(\key, 888);
```

The messages `make(theFunction)` and `use(theFunction)` allow `theFunction` to be evaluated within the name space defined by the receiving `Environment`. The message `make` returns the `Event`, and the message `use` returns the return value of the function. In the following example, an event is made that defines `~freq` in terms of `~note` and `~octave`. That event is then used to compute the frequency when `~note` is 0:

```
a = Event.make {
  ~octave = 5;
  ~freq = {(~note + (~octave * 12)).midicps};
}
a.use {~note = 0; ~freq.value};
```

This is the basic approach used by `Event's` default mechanism. In that context, one could bind patterns either to `\note` and `\octave` or directly to `\freq`. To allow this kind of overriding of the default mechanism, a new copy of the `protoEvent` must be used to generate each event in the sequence.

6.7.2 The Default Parent Event and Event Types

`Event's` default mechanism is implemented through a default parent event stored in the class variable `defaultParentEvent`. This event defines the play functions of different event types and provides values for all the keys they use. Figure 6.16 provides the definition of `Event's` play method, and figure 6.17 provides the default definition of the key `\play` that method uses.

```
play {
   if (parent.isNil) {parent = defaultParentEvent};
```

```
    this.use {~play.value};
}
```

Figure 6.16
The definition of `Event's` play method.

```
{
    var tempo, server;

    ~finish.value;    // user callback
    server = ~server?? {Server.default};
    tempo = ~tempo;   // assigning to a variable

    // saves repeated look ups
    if (tempo.notNil) {// if not nil, change tempo of
         thisThread.clock.tempo = tempo; // the clock   // playing t
he pattern
    };
    ~eventTypes[~type].value(server);   // select play function   fr
om ~type
}
```

Figure 6.17
Definition of the key `\play` in the default event.

Thus the `defaultParentEvent` is used unless the event specifies otherwise, and the function stored at `\play` defines the event's response to play. The final line of that function implements event types.

An event type is simply a play function that receives `server` as an argument. All event types are held in an `Event` stored in `defaultParentEvent[\eventTypes]`. New event types can be added using the class method `*addEventType(key, function)`. Figure 6.18 presents the definition of the event type `bus` as an example.

```
{|server|
    var lag, array;
    lag = ~lag + server.latency;
    array = ~array.asArray;
    server.sendBundle(lag,
         [\c_setn, ~out.asUGenInput, array.size] ++ array);
}
```

Figure 6.18
Implementation of the event type `\bus`.

Notes

1. As noted in chapter 5, `Symbols` are objects that uniquely represent a sequence of alphanumeric characters. The language ensures that there is one, and only one, `Symbol` for any given sequence of characters. Symbols are written enclosed within single quotes or preceded by a backslash (e.g., `'a Symbol with spaces'` or `\aSymbol`).
2. A possible source of confusion is the convention that blocks of code in examples are also enclosed in parentheses. Such parentheses make it possible to select the text of the example with a double click, and they are usually not included in the block of text evaluated by the `Interpreter`.
3. The value returned by `delta` can be set directly by assigning a value to the key `\delta`. This option is used by patterns that run several subpatterns in parallel. It enables them to schedule those subpatterns without altering their internal rhythmic specification (i.e., `\dur` and `\stretch`).
4. Strictly speaking, the value pattern specifies a sequence, and it is that sequence that determines the value of the key to which the pattern is bound.
5. The asterisk causes `Pbind` to treat the elements of the key/value array as individual arguments.
6. There is no direct way to determine the duration of the sounds specified by a pattern. For example, the last event could play a synth with an arbitrarily long decay envelope. Consequently, `render` requires the duration of the sound file to be specified.
7. The pattern `Pproto` and the class `EventTypesWithCleanup` are recent additions to the language that simplify the allocation, deallocation, and use of buffers and buses in patterns. These classes provide a supplemental collection of buffer-related `eventTypes` (`table`, `sine1`, `sine2`, `sine3`, `cheby`, `cue`, and `allocRead`) that perform standard initialization tasks. Consult `Pproto`'s Help file for details.
8. The server-related keys are altered when a pattern is used to generate a score or render an audio file.
9. A similar mechanism provides *multichannel expansion* in SynthDefs.
10. When an event pattern defined by `Pbind` is sent `asStream`, it creates a key/stream array corresponding to the key/value array that defines it. It relays `next(protoEvent)` messages to those streams in order, setting each bound key in the `protoEvent` to the value returned by its stream.

7 Just-in-Time Programming

Julian Rohrhuber and Alberto de Campo

It is impossible to write a program while it runs. A program describes and determines a process, so changing the description implies a new outset, a new process. Yet it is possible to structure a program in such a way that parts of it can be interchanged dynamically, and its textual form can be arranged to allow rewriting those parts while the whole process continues. Thus, instead of first designing an application that has fixed interaction points (parameters), the program text itself becomes the main interface. Instead of planning ahead and providing a large number of parameters, we may modify the program itself at runtime. Although in many fields, the program is of interest only for the desired application, here the program is a reflection of thinking and an integral part of the reasoning process.

This paradigm is relevant for improvised music and performance art since in live coding, artists write algorithmic compositions in the concert and use code as a conversational medium.[1] Also, programming is an integral part of concept formation both in scientific research and in the experimental prototyping of algorithms, so that often a new idea emerges from minor details or misconceptions that, not unlike the fringe of an atoll, turn out to be the eventual solutions.

Sound programming is at the same time a difficult and a captivating activity because of the way abstract expressions and listening experiences relate to each other—the static text structure of code on the one side, the unfolding of sound in time on the other. Programming is not merely the construction of a calculating mechanism to receive the answer to a question, but rather a way to find the right question. Thus, conversational and interactive programming have been undercurrents in computer science and experimental mathematics since the 1960s (Klerer and Reinfelds, 1968), developing further in the form of live coding, operating system design, and scientific experimental programming (e.g., Vogt et al., 2007). Here, we will discuss a class library within SuperCollider (JustInTime Programming Library, or JITLib) that provides such an experimental and improvisational environment.

One interacts with SuperCollider in a manner similar to Smalltalk or Lisp: although conversational approaches, like terminal applications, work more or less on a line-by-

line basis, here, appropriate parts of the program text are selected and evaluated, while the rest of the system is active.

Whether considering programming as a dialogue or as symbiosis with the "thinking machine" (Licklider, 1960), a process of external reasoning (Suchman, 2007; Clark and Chalmers, 1998; Iverson, 1979), an "exploratory object" (Guardans 2010), or the formation of an "epistemic thing" (Rheinberger, 1997), the relation between program text and running program is not trivial; analysis and experiment are interdependent. Thus, in order to be able to change one's mind while changing one's program, one has little choice but to include the programming activity itself in the program's operation.

7.1 Changing State

After showing the basic issues of just-in-time programming by simple examples, we demonstrate how programs can be written as they run. Figure 7.1 shows a small algorithm that multiplies the values of the (interpreter) variables `x` and `y`, determines the remainder after division by 5, and changes *x* to the result (so, e.g., `4 * 13 mod: 5` results in 2).

```
(
Task {
   x = 4; y = 13;
   loop {
        x = (x * y) % 11;
        (note: x.postln, dur: 0.125).play;
        0.125.wait;
   }
}.play
);
// change x and y
x = 5;  // new initial value
y = 4;  // new multiplication factor
```

Figure 7.1
A modulo algorithm that operates over states of variables.

After *x* and *y* are assigned initial values, the algorithm runs within a task that waits a little while after each of these steps. The variables *x* and *y* represent a momentary state of the program, one that is partly changed by the program at runtime. As a consequence, the expression `x = 4; y = 13;` which initialized this state, does not directly stand for these values at runtime: the code represents only the initial moment.

Because the interpreter and our task run concurrently, we may now use the line of code `x = 5` and change either *x* or *y* to a different value while it is being operated on. This change will always fall between one operation and the next, while the task is waiting. The series of resulting numbers proceeds from this new state.

Thus, as long as we have access to a state that is read repeatedly over time, such as the number in this example, modifying code at runtime is simple; once we are concerned with more abstract descriptions of changing or shared state, this is not as trivial: figure 7.2, though similar, uses variables assigned to unit generators. Here sine oscillators, each with its own frequency, are multiplied by each other. The remainder after division by 0.4 is then used, with some scaling, as the frequency input of another sine oscillator. A slightly "creaky" tone results.

In figure 7.2, any attempt to change the value of *x* or *y* fails. One can only stop the whole program, change it, and then rerun it. This is because the variable does not stand for a changing state here but for a unit generator, an abstract description of a whole process rather than of an initial condition. In other words, this description holds for the entire time the process will run, not just at a single moment. As a consequence, it is not trivial to say what it would mean to change this description at a given moment.

```
(
{
   x = SinOsc.kr(4);
   y = SinOsc.kr(13);
   SinOsc.ar(x * y % 0.4 * 500 + 600) * 0.2
}.play;
)

// change x and y?
x = SinOsc.kr(4); // no effect.
y = SinOsc.kr(4); // no effect either.
```

Figure 7.2
Synthesis graph.

To briefly foreshadow a solution, figure 7.3 shows a specific kind of environment, a `ProxySpace`. In this code, each line can be evaluated in any order; there is no longer any difference between the code used for initializing and the code used in rewriting. Furthermore, there is no difference between changing a number, an operator, or a unit generator, since any change in the text is reflected in a change to the running system.

```
p = ProxySpace.push;
~x = {SinOsc.kr(4)};
~y = {SinOsc.kr(13)};
~z = {SinOsc.ar(~x * ~y % 0.4 * 500 + 600) * 0.2};
~z.play;

// now ~x and ~y can be replaced
~x = {SinOsc.kr(0.4)};
~y = {SinOsc.kr(1.3)};

p.clear(2).pop; // release environment (2 sec fade out)
```

Figure 7.3
Dynamic synthesis graph.

To understand the principle behind this type of solution, we give an introduction to a common form of abstract description of processes in SuperCollider.

7.2 Abstraction and Proxies

Many objects in SuperCollider are abstract in the sense that, when operating with them, we do not get a completed value, but rather an algorithm that produces values only when applied. The simple comparison: `x = [1910, 1911, 1912]; y = x + 96.rand;` results in an array of numbers for *y*. On the other hand, an expression such as `y = x + {96.rand}` results in a `BinaryOpFunction`, an object that describes this addition without yet performing it. Unlike the previous example, this function returns a new value not only once, but each time that it is evaluated; the function represents the operation of adding a new random number to *x*. (The result is calculated each time `y.value` is called.) Thus, we have composed a new calculation rather than directly calculating a result once; we have described a description.[2]

In SuperCollider, this type of behavior is common; all the subclasses of `AbstractFunction`, such as `UGen`, `Stream`, and `Pattern`, behave analogously. This is an important feature for working with sound: rather than specify each moment in time directly, we can calculate with sound generators on a more abstract level (figure 7.2 shows such a case), combining them into complex graphs of calculation. In fact, even this combination of UGens and patterns is commonly represented as a program. For instance, the expression

```
{var x = 1.0; 5.do {x = x * SinOsc.ar(1911.rand)}; x} .play
```

specifies a chain of multiplied oscillators which, when instantiated, processes a value continually (i.e., for each sample at the audio rate). Inside a `SynthDef`, this chain can be used to define further processing. SynthDef itself can be used to define any number of synth nodes, which is where the process actually happens.

In other words, abstraction permits the interpretation of math operations and other messages as potential operations or "operations on the future," combinations of possibly multiple future processes. Since these processes are not precalculated, their behavior may depend on external changes such as audio input or some other interaction.

The structures of operations and interconnections may be changed not only via their parameters, but also through rewriting their descriptions (figure 7.3). In SuperCollider, proxies allow such structural changes. Here, a proxy is a placeholder for potential redefinitions. It forms a link between the domain of stateless, functional language and the imperative domain through the action of rewriting (Rohrhuber et al., 2005; or, for comparison with a much earlier approach, Mathews 1969). Using different types of placeholders, program code can be refactored at runtime or even created as an empty skeleton of "roles" to be filled later. (This is essentially an extension of late binding.)

To be compatible with the different processes, a proxy has to behave differently, depending on whether it refers to a server node, a task, a pattern, or an object in general. (The latter is touched upon in chapter 23.) The proxies we will discuss here form a bridge between abstract stateless descriptions of processes such as UGen graphs or patterns and interactive stateful processes unfolding in time. Somewhat similar to free variables, proxies provide schemes of access (in order to reach a description and a running process created from it) and schemes of replacement (in order to change a description and affect its associated process and any processes connected to it).

Usually, free variables have to be bound to some object before they can be used in a process. To allow a proxy to be used in multiple places before its object is known, the proxy schema abstracts from this order of assignment. Creating a proxy and changing its object (its source) may be done explicitly; alternatively, the implicit `def` syntax or a special environment such as `LazyEnvir` or `ProxySpace` allows a syntactical unification of instance creation, reference, and assignment for each type of proxy (e.g., `Ndef(\x)` for reference and `Ndef(\x, {PinkNoise.ar})` for assignment). As a consequence, the same code that created the placeholder instance can be modified and evaluated at any time after the process is first running.

7.3 ProxySpace, Ndef, and NodeProxy

The SuperCollider server allows interconnecting synth processes and exchange nodes at runtime. For efficiency reasons, there is a strict conceptual difference between the graph of units inside a single synth process and the graph between synth processes which

depend on each other. While it is all still SuperCollider language, code that describes synth definitions has a semantics different from the code that is used to interconnect them. For example, Synth objects are meticulously kept outside of SynthDefs, and there is no place for changes to the UGen graph within `Synth` objects. `ProxySpace` is an `Environment` (a set of things that can be accessed by name; see the Environment Help file for more details) that brings processes and interconnections into a more direct relation: hiding some functionality, it makes it easier to rewrite parts of a UGen graph as a synth graph and to combine a synth graph into a single UGen graph.

Figure 7.4 shows refactoring code at runtime: in the upper part, there is a pair of `UGen` graphs (one in `~a` and one in `~b`), which have been rewritten into the equivalent node graph (now with `~a`, `~b`, `~c`, and `~d`). Figure 7.5 visualizes such operations.

```
p = ProxySpace.push;
p.reshaping = \elastic;

~a = {Lag.ar(LFClipNoise.ar(2! 2, 0.5, 0.5), 0.2)};
(
~b = {
   var c, d;
   c = Dust.ar(20! 2);
   d = Decay2.ar(c, 0.01, 0.02, SinOsc.ar(11300));
   d + BPF.ar(c * 5, ~a.ar * 3000 + 1000, 0.1)
}
);

~b.play;

// the refactored code from above

(
~a = {
   var a;
   a = Lag.ar(LFClipNoise.ar(2! 2, 0.5, 0.5), 0.2);
   BPF.ar(~c.ar * 5, a * 3000 + 1000, 0.1)
}
);
~c = {Dust.ar(20! 2)};
~d = {Decay2.ar(~c.ar, 0.01, 0.02, SinOsc.ar(11300))};
~b = ~a + ~d;
~b.play;
```

Figure 7.4
Refactoring a synthesis graph at runtime.

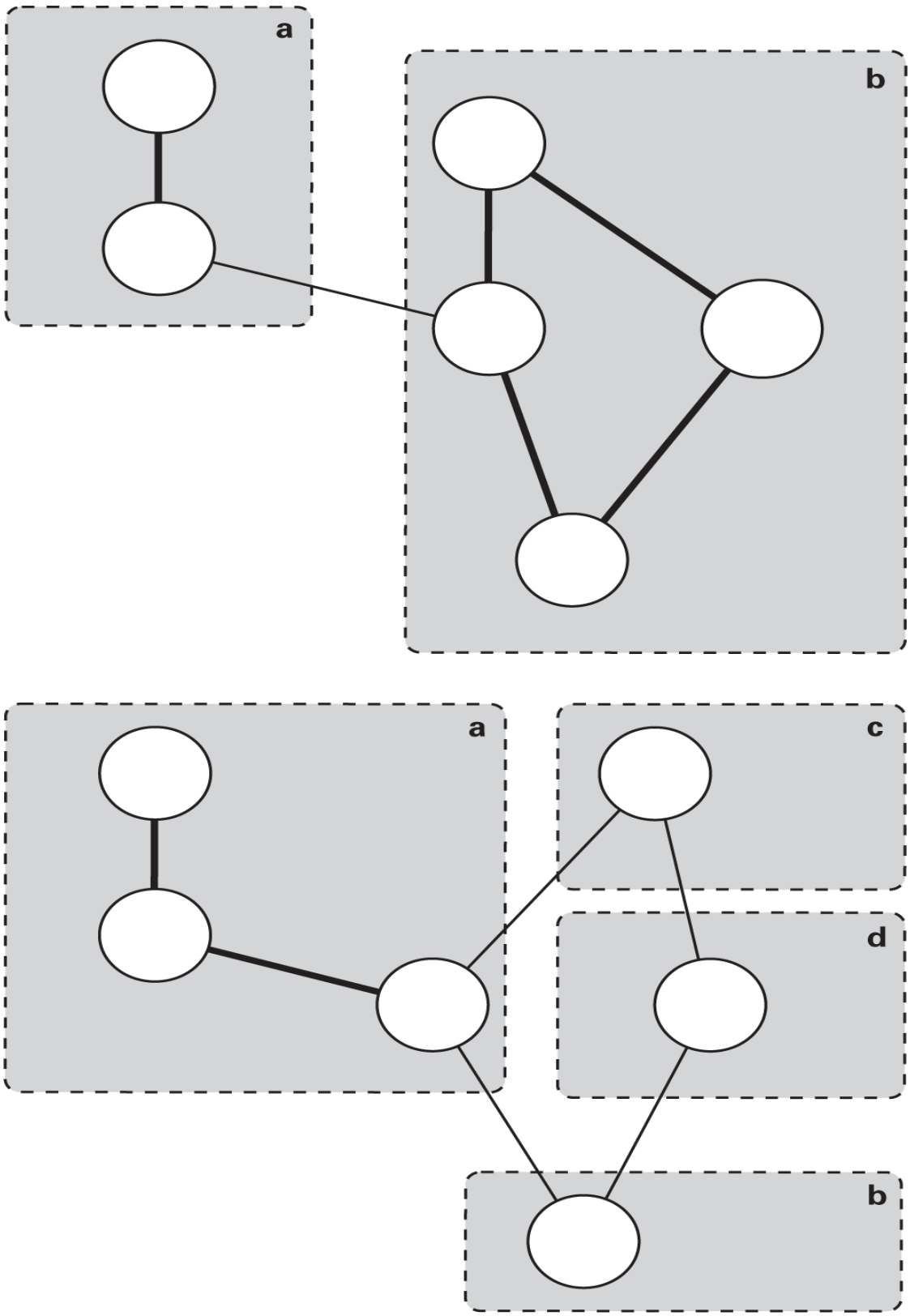

a
b
a
c
d
b
Dynamic binding in synth graph
Static binding in synth graph
Unit generator
Node proxy

Figure 7.5

On the whole, a `ProxySpace` behaves like any other environment: externally, it can be accessed via messages such as `put` and `at`; from within, the tilde (`~`) can instead be written. Although environments normally represent objects on the client side, `ProxySpace` represents descriptions and running instances of synthesis processes on the server.

Whenever it is accessed, `ProxySpace` returns a placeholder for a server node—instances of `NodeProxy`. Thus, assigning a value to the environment (e.g., `~x = [1, 2]`) results not only in keeping this value but also in its conversion into a `UGen` graph, and a corresponding synthesis process on the server which the `NodeProxy` takes care of. Without any reference to bus numbers or execution order, each node is addressed only by name; thus, one may reason about synthesis from a symbolic point of view.[3] Each symbol represents a place for a synth process that can be accessed from any other synth process on the same server (even from within itself: `~x = ~x + ~y % 11` is a recursive and slightly chaotic function; see also figure 7.6). When a placeholder is used inside a `UGen` graph, it can be employed empty (returning silence) and may be filled later (e.g., `~x.play; ~x = ~y * ~z`). By abstracting away execution order, it is now easy to use a piece of program text not only as a single block, but also in parts that can be evaluated either together or apart, thus accessing and changing a sound aspect by aspect. By itself, the synthesis graph has no privileged observer position. A node becomes audible only when explicitly playing it through a hardware output channel (`~x.play`), so any audio rate proxy can be listened to equally, perhaps shifting the center of attention from one to the next. On multichannel systems, the `playN` variant allows a further method of monitoring a `NodeProxy`: mapping multiple outputs to multiple channels (see the `playN` help).

```
// self reference (~x) constructs a loop at control rate

~out.play;
~x = 0.2; ~a = 1.1; ~c = 0.13;
~x = (~a * ~x) + ~c % 1.0;
~out = {SinOsc.ar(~x.ar * 4000 + 200) * 0.1};
```

Figure 7.6
A dynamic graph of a chaotic linear congruence.

So far, we have used `ProxySpace` "from within" for simplicity, employing `p = ProxySpace.push` to replace the current environment with this specific environment

temporarily. To return to the usual mode, `p.pop` is evaluated; of course, we may also create a proxy space without pushing it (`p = ProxySpace.new`). So if one prefers to use the default environment instead of variables, it is as valid to write `~x =…` as it is to write `p[\x] =.…` Furthermore, `~x.ar` is equivalent to `p[\x].ar`. As an overview, figures 7.7–7.9 show the same sound expressed in different ways—inside other classes, it is usually better to create an object instance, whereas for the development of a network of interconnected nodes, either `Ndef` or `ProxySpace` may be more appropriate. The same scheme applies to the other proxies, which we will discuss further next, such as `PatternProxy` (`Pdefn`, `Pdef`) and `TaskProxy` (`Tdef`), and to `LazyEnvir`, which has a function similar to `ProxySpace`.

```
n = NodeProxy.new;
x = {SinOsc.ar(n.kr * 200 + 300) * 0.1} .play;
n.source = {LFPulse.kr([1.3, 2.1, 3.2]).sum};
n.clear; x.free;
```

Figure 7.7
Creating a Proxy object explicitly and changing its source.

```
Ndef(\out, {SinOsc.ar(Ndef.kr(\x) * 200 + 300) * 0.1}) .play;
Ndef(\x, {LFPulse.kr([1.3, 2.1, 3.2]).sum});
Ndef.clear;
```

Figure 7.8
Unified creation and access syntax with `Ndef`.

```
p = ProxySpace.push; // if needed
p.reshaping = \elastic;
~out = {SinOsc.ar(~x.kr * 200 + 300) * 0.1}
~out.play;
~x = {LFPulse.kr([1.3, 2.1, 3.2]).sum};
p.clear.pop;
```

Figure 7.9
Unified creation and access syntax within an environment.

Because `UGen` graphs always have a fixed rate and number of channels, a node proxy, unlike other proxies, has to be initialized to this format. This can be done either explicitly, before it is "filled" with something, or implicitly, derived from what is assigned to it first; monitoring a neutral proxy (`x.play`) will normally initialize it to two audio channels (see figure 7.10; for more details, see also the_lazy_proxy.help). If

required, the message `clear` neutralizes the proxy again so it is free for reassignment. A proxy can be brought into a particular shape while keeping its source (`x.mold`). Also, when we set `reshaping = \elastic`, both the rate and number of channels adjust dynamically to changes in the code, rebuilding other child proxies if necessary (see figure 7.10 and NodeProxy help, subsection Reshaping). Finally, when a `fadeTime` is specified, a slow transition maintains perceptual continuity (see also section 7.7); interesting timbral shifts can be achieved with different times for different proxies. At the control rate, this is a linear, and at the audio rate, an equal power transition.

```
p.clear; // neutralize space, free all buses and synths
p.push; // if needed
~a.ar(3); // 3 channels, audio rate
~b.kr(8); // 8 channels, control rate
~c.play; // playing an uninitialized proxy assumes (per default)
 2 channels, audio rate
~d = {LFNoise0.kr([1, 1, 1, 1])}; // 4 channels, control rate
~a.mold(5, \control); // reshape to 5 channel control rate
~a = {SinOsc.ar(440)}; // reshape with source to 1 channel audio
```

Figure 7.10
Initialization of node proxies in the proxy space.

In SuperCollider, running synth nodes may have controls representing their parameters which can be mapped to control rate buses (e.g., `x = Synth(\default); x.set(\freq, 210)`). A node proxy provides access to these controls in a very similar way, and they can already be set before any synth is present. As soon as a source is given and a synth is created from it, this synth's parameters are set or mapped accordingly (e.g., `~x.set(\freq, 210)`, or mapped to another proxy `~x.set(\freq, ~z)`). Through this, networks of control or audio rate proxies can be changed more efficiently than by reassigning the source (figure 7.11). The additional messages `xset` and `xmap` allow cross-fading parameter transitions, and the operators <<> and <>> connect a series of proxies through their control inputs.

```
~out.play; ~out.fadeTime = 3;
(
~out = {|freq=440, mod=0.4, detune=0.1|
   var in = \in.ar(0!2); // 2 channel audio rate input
   freq = freq * ([0, detune] + 1);
   LFTri.ar(LFTri.ar(mod * freq).range(freq * mod, freq)) * in   *
 0.2
}
```

```
);

~mod2 = {LFNoise1.kr(1).range(0, 1)};
~mod1 = {LFPulse.kr(~mod2.kr * 30 + 1, 0, 0.3)};
~freq1 = {~mod1.kr * 13100 + 100};
~freq2 = {LFTri.kr(30) * 200 + 300};
~audio1 = {BrownNoise.ar(LFClipNoise.kr(10.dup), 1)};
~audio2 = {SinOsc.ar(LFNoise2.kr(1.dup).exprange(4, 1000))};

~out.map(\freq, ~freq2, \mod, ~mod1);
~out.set(\detune, 0.01); // adc: near inaudible here, better   one
 line up?
~out.map(\freq, ~freq1, \mod, ~mod1);
~out.xmap(\freq, ~freq1, \mod, ~mod2); // xmap crossfades over   fa
de time to new value.
~out.xmap(\freq, ~freq2, \mod, ~mod1, \in, ~audio2);
~out.map(\in, ~audio1);
~out <<> ~audio1; // the same as map(\in, ~audio1)
```

Figure 7.11
Parameter mapping and setting.

7.4 Structured Waiting and Rewriting: TaskProxy and Tdef

In sound programming, one typically experiments by evaluating small pieces of code. Once they are understood properly, they can be incorporated into an automatic sequence which replaces the programmer's actions in time and in turn becomes the basis for further modification. On the most basic level, tasks are a way to structure sequences of such actions; in essence, they are time streams of structured waiting. Task proxies provide an interface that makes such tasks easily accessible and rewritable, and wherever combinable subtasks are needed, they provide the necessary abstraction. For instance, in order to change code in the running algorithm shown in figure 7.12, one can arbitrarily modify either the `SynthDef` or the `Tdef` at runtime. This allows for variables local to the process, in contrast to the very first example, enabling any part of the code to be altered without prior reorganization.

```
// this synthdef is used in the subsequent figures
(
SynthDef(\wave, {|out, freq=440, amp=0.1, sustain=0.1, mod=0.2|
   OffsetOut.ar(out,
        EnvGen.ar(Env.perc(ExpRand(0.001, 0.05), sustain, amp),   d
oneAction: 2)
```

```
        *
        SinOsc.ar(freq, SinOsc.ar(sustain.reciprocal * 8,   [0, Ran
d(0, pi)], mod))
   )
}).add
);
(
Tdef(\x, {
   x = 4; y = 13;
   loop {
        x = (x * y) % 11;
        (instrument: \wave, note: x.postln, sustain: 0.5,   octave:
6).play;
        0.125.wait;
   }
}).play
);
```

Figure 7.12
Rewriting a synth def and a task def while running.

Like other proxies, a `Tdef` may be implicitly created and played without any content (e.g., `Tdef(\x).play`) or be accessed like any other object (`x = TaskProxy.new; x.play`) with no change of functionality (analogous to figures 7.7–7.9). Using the `embed` and `fork` messages, several `Tdefs` can be combined: while any waiting done in the embedded task will cause the outer task to wait, the `Tdef` does not wait for a forked task to finish and will play in parallel. Figure 7.13 shows how to branch tasks into parallel streams. In both cases, changes in the `Tdef` itself are threaded into each running process, so that infinite sequences remain responsive to modification.

```
(
Tdef(\a, {10.do {(instrument: \wave, freq: 50.rand + 1500).   play;
0.03.wait}});
Tdef(\b, {[1, 5, 1, 2, 8, 4, 12].do {|x| (instrument: \wave,   not
e: x + 8).play; 0.1.wait}});
Tdef(\c, {"c is just a waiting message".postln; 2.wait;});

Tdef(\x, {
   loop {
        Tdef(\a).embed; // play in sequence
        1.wait;
        Tdef(\b).embed;
        2.wait;
```

```
        Tdef(\a).fork; // play in parallel
        Tdef(\b).fork;
        Tdef(\c).embed;
    }
}).play
);
// rewrite with infinite loop
Tdef(\a, {inf.do {(instrument: \wave, freq: 50.rand + 500).play;
 0.1.wait}});
// rewrite with finite loop
Tdef(\a, {10.do {(instrument: \wave, freq: 50.rand + 500).play;
 0.1.wait}});
```

Figure 7.13
Embedding and forking of different tasks.

Environments may be passed to the forked and embedded tasks to influence each of their behaviors (figure 7.14).

```
(
Tdef(\a, {|in|
    in.at(\n).do {|i|
        in = (instrument: \wave, detune: 5.rand2).putAll(in);
        in.postln.play;
        in.delta.wait;
    }
})
);
(
Tdef(\x, {|inevent|
    loop {
        Tdef(\a).embed((note: [15, 17], dur: 0.01, n: 13));
        1.wait;
        Tdef(\a).embed((note: 9, dur: 0.4, n: 4));
        1.wait;
    }
}).play;
)
```

Figure 7.14
Passing an environment into a task proxy when embedding.

7.5 Empty Patterns

Like unit generators, patterns are stateless descriptions of processes. However, unlike unit generators, networks of patterns specify streams of objects in general, or streams of events, that run within an instance of a clock in `sclang`. Because patterns are only descriptions of processes, the same pattern can be the source of a variety of derivations: other patterns are derived, each of which may serve as a description for any number of streams. These patterns should be entirely independent of generated streams so that their code can be guaranteed to express the process they produce. Thus, no stream should modify its pattern, nor should the pattern know about any of its streams.

This clear separation demands careful implementation of a just-in-time approach. We want to be able to define a neutral placeholder not for a stream (which would be simple), but for such an abstract description. A suitable implementation has a number of requirements: first, a pattern proxy should be usable before its content is known, as shown in the empty Tdef above. Second, the proxy's description should be amenable to rewriting at any point. And third, changes caused by any such rewriting should affect all processes specified by the pattern proxy. To achieve this, the subclasses of `PatternProxy` insert points of interaction into the abstract layer so that a change of its description is threaded into the system at runtime. For object streams such as numerical patterns (e.g., `Pseq([0, 2, 3])`), `PatternProxy` and `Pdefn` provide such an interface; for event streams (e.g., `Pbind(\note, 4)`), `EventPatternProxy` and `Pdef` are interaction points in a network of streams.

Figure 7.15 shows an instance of an object stream in which an instance of `Pdefn` is used directly as the source of a stream of numerical values. Figure 7.16 uses the same definitions to create a stream of arrays from variations of this definition. Using math operations on the placeholder, we are able to derive new patterns that represent independent yet rewritable calculations with processes that will happen in the future. This type of late binding is a typical feature of patterns and streams. An empty `Pdefn` (or, equivalently, `PatternProxy`) embeds a series of integer 1 into the stream, a value that, although it is as generic as 0, is safer for use in time streams (an endless stream of wait times of 0 is a guaranteed program lockup). Any object or pattern that returns objects can fill the source of a pattern proxy, so one may calculate with streams of functions just as well as with streams of numbers; a single value will be streamed out indefinitely (e.g., `Pdefn(\x, [5, 3, 2]); Pdefn(\x).asStream.next`), until it is replaced by something else.

```
Pdefn(\x, Pseq([0, 2, 0, 7, 6, 5, 4, 3], inf));
(
Task {
    var stream = Pdefn(\x).asStream;
    var val;
    loop {
```

```
        val = stream.next;
        (instrument: \wave, note: val).play;
        0.2.wait
    }
}.play
);
Pdefn(\x, Pseq([0, 2, 0, 8, 6, 5, 2, 3, 4, 5], inf));   // rewrite
 the definition at runtime.
Pdefn(\x, Pseq([0, 2, 0, 7, 6, 5, 4, 3].scramble + 4, inf));
```

Figure 7.15
A pattern proxy as an entry point into a stream.

```
Pdefn(\y, Pdefn(\x) + 2); // derive a transposition
Pdefn(\z, Pdefn(\x) + Pseq([0, 5, 0, 7, 2], inf)); // derive   a va
riation
Pdefn(\a, Ptuple([Pdefn(\y), Pdefn(\z)])); // combine them in   a s
tream of arrays
(
Task {
    var stream = Pdefn(\a).asStream;
    var val;
    loop {
        val = stream.next.postln;
        (instrument: \wave, note: val, sustain: rrand(0.5, 0.9)).
play;
        0.2.wait
    }
}.play
);

// rewriting the definitions causes all derivations to vary
Pdefn(\x, Pseq([0, 11], inf));
Pdefn(\x, Pseq([0, 2, 0, 7, 6, 5, 4, 3].scramble + 5, inf));
Pdefn(\z, Pdefn(\x) + Pseq([1, 5, 1, 11, 1], inf)); // change   a v
ariation
Pdefn(\a, 5); // a number as a source
Pdefn.clear; // clearing all—the empty pattern returns a series   o
f 1.
```

Figure 7.16
Deriving variations fom nonexisting streams by mathematical operations.

Event streams behave analogously to object streams shown here, but with the following differences: an event stream may create synth or group nodes on the server, set their controls, and even embed filtering synths into each other. For this slightly different kind of stream, the third pair of proxies discussed here forms an abstract placeholder: `EventPatternProxy` and its named equivalent `Pdef`. Calculating with events, however, is a different matter from doing so using the numerical mathematics on object streams shown above. Since events of some specification are always required, an empty proxy returns a silent event (`Event.silent`) until further modified, and when a given definition is rewritten, the stream takes care of releasing any synths within it. `Pchain` and `Pbindf` are used to combine the event patterns, and their use also enables the composition of sequences of parallel or subsequent patterns (for a `Pbind`-like but incremental pattern definition, see the `Pbindef` Help file, which takes arbitrary pairs of keys and values and implicitly converts them to pattern proxies).

In addition to their placeholder functionality, event stream proxies, like task proxies, provide the option to play one stream internally. As shown in figure 7.17, we can implicitly create a pattern proxy and play, stop, pause, and resume it via the `def` interface, just as we can with `Tdef`. Generally, a `Pdef` may take the place of any event or event pattern, so just like synth graphs with node proxies, networks of patterns can be refactored at runtime (figure 7.18). Further below, we will show how to combine these two worlds.

```
Pdef(\a).play; // play silence in sequence
Pdef(\a, Pbind(\instrument, \wave)); // insert a sequence of   note
s
Pdef(\a, Pbind(\instrument, \wave, \dur, Pseq([1, 3, 2, 3],   inf)
 / 6)); // add some rhythm
Pdef(\a).pause;
Pdef(\a).resume;
Pdef(\a).stop;
```

Figure 7.17
`Pdef`—play, pause, and resume.

```
(
(
Pdef(\x,
   Pbind(
        \instrument, \wave,
        \mod, Pseq([1, 0, 1, 0], inf),
        \dur, Pn(1/2, 8),
        \note, 7
```

```
    )
)
);

(
Pdef(\y,
    Pbindf(
        Pdef(\x),
        \amp, 0.2,
        \note, Pshuf([0, 2, 3, 5], 2) + Prand([0, 5, [0, 4]],   in
f),
        \dur, Pseq([1, 3, 2, 3], inf) / 6
    )
)
);

(
Pdef(\z, Pbindf(Pdef(\y), \dur, 1/4))
);

// the combination of all placeholders into a new placeholder
(
Pdef(\a,
    Pmul(\dur, Pwhite(-0.02, 0.02) + 1,
        Pseq([
            Ppar([Pdef(\x), Pdef(\y)]),
            Pdef(\x),
            Pdef(\y),
            Pdef(\z),
            Ppar([Pdef(\x), Pbindf(Pdef(\y), \ctranspose, 2)])
        ], inf)
    )
);
)
)

// listen to each Pdef on its own:
Pdef(\x).play;
Pdef(\y).play;
Pdef(\z).play;

// listen to the combination:
Pdef(\a).play;
```

```
// go into a looping vamp
(
Pdef(\x,
   Pbind(
        \instrument, \wave,
        \dur, Pseq([1, 3, 2, Prand([3, 2])], inf) / 6,
        \octave, [6, 4]
   )
)
);

// release the break
(
Pdef(\x,
   Pbind(
        \instrument, \wave,
        \dur, Pseq([1, 3, 2, Prand([3, 2])], 1) / 6,
        \octave, [6, 4]
   )
)
);

Pdef(\a).stop; // stop the player
```

Figure 7.18
A larger combination of `Pdefs`.

7.6 Symbol Streams and Recursive Patterns

A structure with a large number of pattern placeholders can look more complex than necessary, in particular the combination of parallel and serial streams. A simple pair of classes is useful here: `Psym` (for event streams) and `Pnsym` (for object or number streams). Every `Pdef` (or `Pdefn`, respectively) can be reached by a pattern of symbols passed to this pattern. Thus, the sequence `Pseq([Pdef(\x), Pdef(\y), Pdef(\z)])` becomes `Psym(Pseq([\x, \y, \z]))`, and the partly parallel sequence `Pseq([Pdef(\x), Ppar([Pdef(\y), Pdef(\z)])])` can be simplified as `Psym(Pseq([\x, [\y, \z]]))`; see figure 7.19.

```
// the combination of all placeholders into a new placeholder
(
Pdef(\b, Pbindf(Pdef(\y), \ctranspose, 2));
Pdef(\a,
   Pmul(\dur, Pwhite(-0.02, 0.02) + 1,
```

```
            Psym(Pseq([[\x, \y], \x, \y, \z, [\x, \b]], inf).trace)
    // trace it to post which
      )
).play;
)
```

Figure 7.19
Simplifying the code in figure 7.18 using `Psym`.

It is much easier to restructure musical material in such a way, bringing serial and parallel approaches syntactically closer to each other. Furthermore, when using a `Pdefn` for the sequence of symbols, the composition of the whole piece may be rearranged hierarchically (see figure 7.20).

```
(
Pdefn(\sequence, Pseq([[\x, \y], \x, \y, \z, [\x, \b]], inf));
Pdef(\a,
    Pmul(\dur, Pwhite(-0.02, 0.02) + 1,
         Psym(Pdefn(\sequence).trace)
    )
).play;
)

// rewrite the sequence
Pdefn(\sequence, Pseq([\x], inf));
Pdefn(\sequence, Pseq([\x, \y, \x, [\x, \y]], inf));

Pdef(\a).stop; // stop playing
```

Figure 7.20
Using `Pdefn` for the sequence of symbols itself.

With the `Pnsym` pattern, the numerical stream variant works analogously, except that instead of expanding in the form of a `Ppar`, it does so in the form of a `Ptuple`, returning arrays of numbers or objects.[4] Note that as an alternative, `Psym` and `Pnsym` can be given a dictionary with patterns that they use as a lookup, so that they can be used independently of `Pdef` and `Pdefn`.

A second type of parallelism, resembling the branching of a tree rather than the lanes of the highway next to it, is implemented in a specific event type, named phrase. When such an event is played, instead of sending a single sound event to the server, the event retrieves a `Pdef` matching its instrument key (alternatively, from a repository given in the event itself). The event then plays a phrase—a stream of events—from this template, applying the event's sustain value as a temporal limit. Consequently, `(instrument:`

`\x, type: \phrase, sustain: 3).play` will play a phrase (given that `Pdef(\x)` has been defined in one of these examples), and `Pbind(\type, \phrase, \instrument, \x, \legato, 1.5).play` will play an overlapping sequence of these phrases (for an alternative, see the `Pspawn` and `Pspawner` Help file). Taking advantage of the syntactical similarity of `Pdef` and `SynthDef`, a `Pdef` may be written, and thought of, as a parameterized phrase generator (figures 7.21–7.22). Here, in common with a synth definition, the arguments supplied by the function specify the sound generation, but in the case of `Pdef`, it occurs on a phrase-by-phrase basis. As a result, as each event from the outer pattern is passed in, its values may be used for further processing, and even for creating other phrase patterns in turn. Polyphonic events branch into their respective subphrases.

```
(
Pdef(\x,
   Pbind(
        \instrument, \wave,
        \dur, Pseq([1, 3, 2, Prand([3, 2])], 1) / 6,
        \octave, [6, 4]
   )
)
);
(instrument: \x, type: \phrase).play; // a single phrase from   Pde
f(\x)

Pdef(\x).playOnce; // or equivalently, play Pdef(\x) once

// a pattern of overlapping phrases
(
Pbind(
   \type, \phrase,
        \instrument, \x,
        \legato, 2.5,
        \dur, Pseq([1/3, 1.5], inf),
        \note, Pseq([0, 5, 8], inf)
).trace.play
);
```

Figure 7.21
`Event` type "phrase".

```
(
Pdef(\x, {|note=0, n=6, step=4, modulo=15, sustain=1|
   Pbind(
```

```
            \instrument, \wave,
            \note, note.value + (Pseries(1, step, n) % modulo) + 7,
            \dur, sustain.value / n
        )
    })
    );

    (
    Pdef(\a,
        Pbind(
            \type, \phrase,
            \instrument, \x,
            \note, Pseq([0, 5, 4, 8, 0], inf),
            \n, 5,
            \modulo, Pseq([3, [4, 3, 5], [13, 15]], inf),
            \dur, Pseq([1, 2, 0.5, 1.5], inf)
        )
    ).trace.play
    );

    (
    Pdef(\a,
        Pbind(
            \type, \phrase,
            \recursionLevel, 1,
            \instrument, \x,
            \note, Pseq([0, 5, 4, 8, 0], inf),
            \n, 5,
            \modulo, Prand([3, [4, 3, 5], [13, 15]], inf),
            \dur, Pseq([1, 2, 0.5, 1.5], inf) * 2
        )
    ).play
    );
```

Figure 7.22
Recursive phrasing.

The event type phrase has a number of parameters that may be used to extend this behavior and to compose recursive phrases: if a numerical `recursionLevel` is provided, the branching of a subpattern does not stop at the first level of replacement but continues recursively to the depth specified (for a detailed discussion see the help file recursive_phrasing).

7.7 Perceptual Continuity and Context

Within programming experiments and live coding sessions, changes that interfere with the algorithmic processes and restructure their rules are part of one and the same musical stream. A rupture introduced as part of the algorithm and one that stems from a rewriting of the algorithm may sometimes be clearly distinguishable, while in other cases, minor modifications remain subliminal. Sonic research is a field in which the relation between an algorithm and a sound can be likened to a measuring instrument. Here, just as in learning sound synthesis, differential hearing becomes an essential method: in order to understand what a program does, minor changes to the code are superimposed upon the changes in the result. The perceptual continuity required may be achieved by a context of processes that remain unchanged. The persistence of parameter settings, and cross-fading or interpolation between old and new versions, contribute to this continuity. Finally, the exact timing of the modification may also be important, for instance, when rewriting trigger algorithms).

In the last part of section 7.3, we showed how to set parameters for a `NodeProxy` that are applied to each new synth created within it. Using the equivalent `Ndef` syntax, we can write, for instance, `Ndef(\x).set(\freq, 367)`. When we add a synthesis function to this code (e.g., `Ndef(\x, {|freq| SinOsc.ar(freq) * 0.1}))`, the frequency of the sine oscillator is adapted to this context. A different function may interpret this context entirely differently.

Earlier in the chapter, we also noted that setting a quantization (`quant`) value (`Ndef(\x).quant =…`) allows changes to be scheduled sample-accurately to a clock's beat, and by supplying a `fadeTime`, a continuous transition is achieved. All pattern proxies (i.e., `Tdef`, `Pdefn`, `Pdef`, and their respective base classes) support both a similar syntax for setting the context of a process and an extensible scheme of exactly when a replacement becomes effective.

Pattern proxies have an optional variable for a context, an environment that is implicitly created if need be. This environment keeps variables across changes of the proxy source and is reachable from within the latter. Using the `set` message, the pair of key and value in this environment can be set or may be directly accessed by the `envir` getter. In the case of `Pdefn` and `Tdef`, this environment is passed as an argument to any function provided there (see also figure 7.14), whereas with `Pdef`, the environment is chained into the event stream. A `Pdef` may also be given a fade time, which then cross-fades between old and new streams.

To specify when the `fadeTime` transition should happen, a number of parameters come into play. First, there is a quant value that works as elsewhere in SC: a number specifies the quantization and an array specifies the triple of `quant`, `phase`, and `timingOffset` (see the `Quant` Help file). Furthermore, pattern proxies understand a fourth value, the `onset`, which, when given, helps to modify longer sequences interactively without reset. Second, a condition function can be provided, which acts to

defer any change until the current stream value fulfills it; thus, for instance, specific insertion points into a melody may be defined.

Although just-in-time programming is very convenient for the composition of algorithms and for keeping code present in mind, it is less ideal for tasks such as mixing or setting parameters. To overcome this problem, a number of loosely coupled GUI classes accompany the different proxies (figures 7.23 and 7.24): `PdefGui`, `TdefGui`, `NdefGui`. In particular, they provide an alternative way of accessing and overseeing playing state involving parameters such as node settings, mappings, and stream environment variables. Drawing on experience with live coding performance situations, the use of these GUIs combines typical tasks such as volume adjustment and quick editing of source code, and it is designed to be completely independent of the rest of the system (throwaway GUIs), which can be used in custom GUIs. The classes `PdefAllGui`, `TdefAllGui`, `NdefMixer`, and `ProxyMixer` provide an overview of all current proxies.

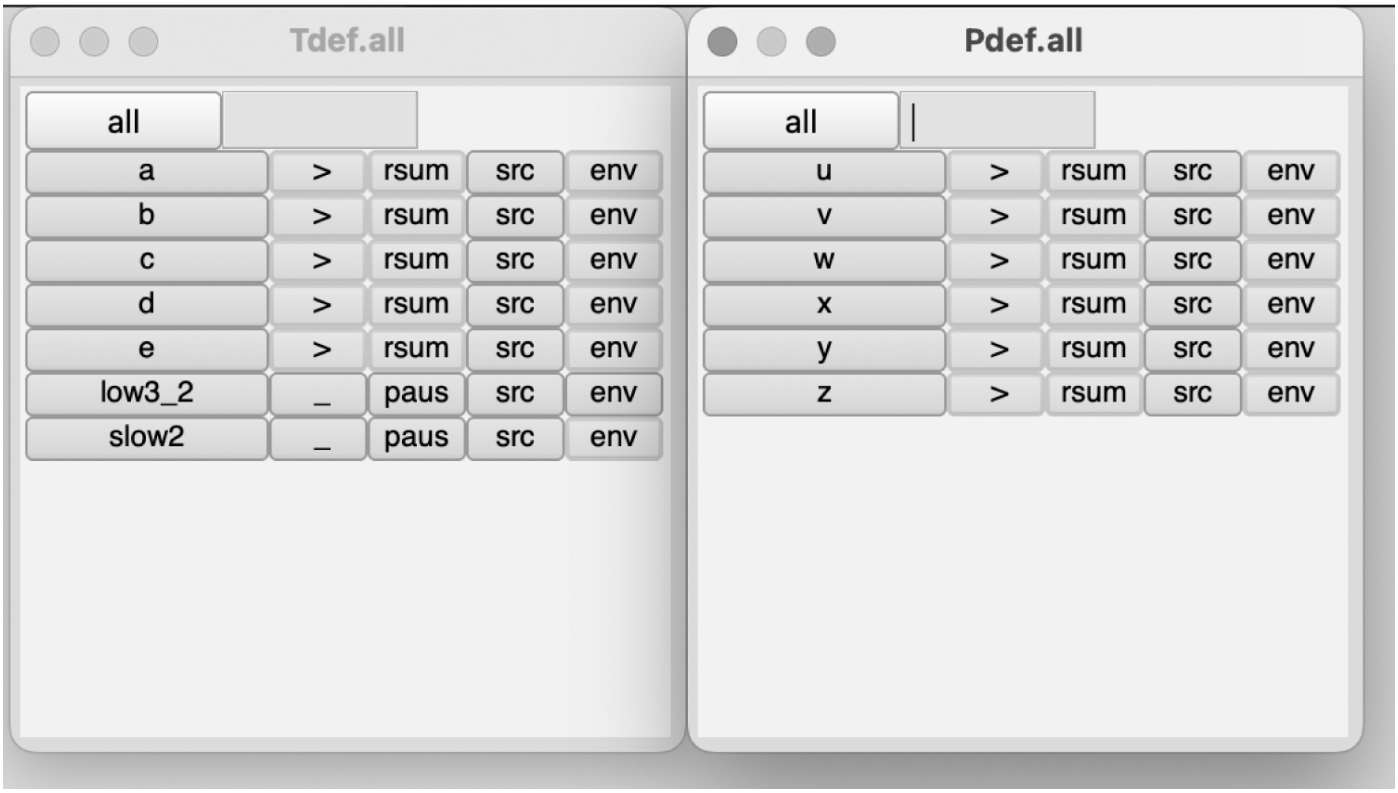


Figure 7.23

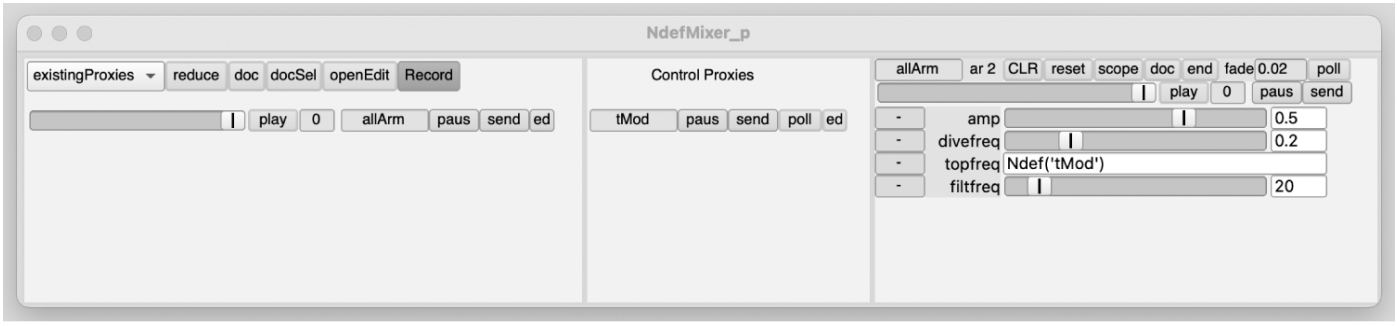


Figure 7.24

7.8 Combinatorics and Extensions

To finally combine all those different types of processes, one may use proxies as adapters. Many kinds of objects can be a source for a `NodeProxy`; they are converted internally so that their synths play on the appropriate bus and group. While a pattern proxy may be used to rewrite patterns that occur in multiple places, a `NodeProxy` can host an event stream, represented by a pattern (or, of course, a pattern proxy, which is just another pattern). Thus, in figure 7.25 the stream plays in `~x` and is filtered by `~y` and `~z` (as usual, `Ndef(\x)`, `Ndef(\y)`, and `Ndef(\z)` would behave equivalently; see figure 7.26).

```
(
SynthDef(\train, {|out, xfreq=15, sustain=1.0, amp=0.1, pan|
   Line.ar(1, 1, sustain, doneAction:2);
   OffsetOut.ar(out, Pan2.ar(Impulse.ar(xfreq), pan, amp));
}).add
);

p = ProxySpace.push;
~z.play;

// A pattern in an audio rate node proxy. . .
(
~x = Pbind(
   \instrument, \train,
   \xfreq, Pseq([50, Pwhite(30, 800, 1), 5, 14, 19], inf),   // onl
y non-standard keys, i.e. xfreq
   \sustain, Pseq([Pwhite(0.01, 0.1, 1), 0.1, 1, 0.5, 0.5], inf),
   \pan, Prand([-1, 1], inf) * 0.1
)
);
~y = {Ringz.ar(~x.ar, 5000 * [1, 1.2], 0.01)}; // resonant   filter
```

```
on the impulses from ~x
~mod = {LFNoise1.kr(0.1).exprange(200, 5000)}; // a modulator
~z = {~y.ar * (SinOsc.ar(~mod.kr) + 1)}; // ring modulation   with
 frequency ~mod

//. . .and a control rate node proxy in a pattern.
// To pass on modulation like this, the standard event   parameters
like freq cannot be used.
// Here, we use xfreq instead.

(
~x = Pbind(
   \instrument, \train,
   \xfreq, Pseq([50, ~mod, 5, ~mod, 19], inf), // read from the   ~
mod proxy bus.
   \sustain, Pseq([Pwhite(0.01, 0.1, 1), 0.1, 1, 0.5, 0.5], inf),
   \pan, Prand([−1, 1], inf) * 0.1
)
);
```

Figure 7.25
Combinations between patterns and UGen graphs.

```
(
SynthDef(\train, {|out, xfreq=15, sustain=1.0, amp=0.1, pan|
   Line.ar(1, 1, sustain, doneAction:2);
   OffsetOut.ar(out, Pan2.ar(Impulse.ar(xfreq), pan, amp));
}).add
);

Ndef(\z).play;

// a pattern in an audio rate node proxy. . .
(
Ndef(\x, Pbind(
   \instrument, \train,
   \xfreq, Pseq([50, Pwhite(30, 800, 1), 5, 14, 19], inf),
   \sustain, Pseq([Pwhite(0.01, 0.1, 1), 0.1, 1, 0.5, 0.5], inf),
   \pan, Prand([−1, 1], inf) * 0.1
))
);
Ndef(\y, {Ringz.ar(Ndef(\x).ar, 5000 * [1, 1.2], 0.01)});
Ndef(\mod, {LFNoise1.kr(0.1).exprange(200, 5000)});
```

```
Ndef(\z, {Ndef(\y).ar * (SinOsc.ar(Ndef(\mod).kr) + 1)});   // ring
modulation with Ndef(\mod)
//. . .and a control rate node proxy in a pattern

(
Ndef(\x, Pbind(
   \instrument, \train,
   \xfreq, Pseq([50, Ndef(\mod), 5, Ndef(\mod), 19], inf),   // rea
d from the Ndef(\mod) proxy bus
   \sustain, Pseq([Pwhite(0.01, 0.1, 1), 0.1, 1, 0.5, 0.5], inf),
   \pan, Prand([-1, 1], inf) * 0.1
))
);
```

Figure 7.26
The same functionality, using `Ndef` instead of `ProxySpace`.

The output of a control rate node proxy may also be mapped to values in a running stream; in the second part of this code example, the modulator proxy (containing an `LFNoise` process) is mapped to a number of the synths spawned by `~x`.

Another typical practice is to use `Tdefs` to spawn synth processes. This can be done as shown in section 7.4, but Tdefs may also interact with other proxies. Setting a `NodeProxy` source at a high speed is efficient only when using precompiled `SynthDefs` (see jitlib_efficiency.help), but by using the `send` message, the proxy can become a placeholder for multiple synths from a single definition, as shown in figure 7.27.

```
Ndef(\x).play; // here an Ndef is used, the same can be done   with
in a ProxySpace
Ndef(\x, {|freq=5, detune=0| Impulse.ar(freq * [detune, 1-detune])
* 0.2});
(
Tdef(\c, {
   loop {
        Ndef(\x).fadeTime = rrand(0.1, 3.5);
        Ndef(\x).send([\freq, exprand(2, 400), \detune, [0.0,   1.
0].choose.rand]);
        2.wait;
   }
}).play
);
```

Figure 7.27
Using `Tdef` to create overlapping synths within a node proxy.

Even though designed for experimenting with flexibility, proxies have turned out to be both useful and reliable building blocks for larger architectures, where they keep the possibilities for major redesign decisions open permanently. In such projects, changes of direction can happen whenever a different perspective seems promising to try.

Since the SuperCollider live coding experiments in the late 1990s, the development of the idea of runtime programmeability has spread and become a desirable feature for systems in general. The approach is now commonly used within SuperCollider for a wide variety of projects. The Quark JITLibExtensions contains useful classes for interactive compositions, installations, performance setups, and open-ended experimental systems. Also, there are a number of libraries that extend its basic principles, be it for live coding as performance or for adapting its approach to specific situations. Many of these come in Quarks, such as NTMI or ddwChucklib-livecode. Quarks like HyperDisCo and Utopia support network music and synchronous programming in ensembles. Because it is such a central aspect of live coding, we end this chapter with collective thinking.

7.9 Networked Live Coding and Recursive History

After devoting much space to detailed examination of proxy behaviors, we end this chapter with a broader discussion. First, the proxy system can be seen as an interaction point not only for a synthesis process, but also between persons. Second, interaction leaves a trace, a history of changes that can mirror future possibilities.

The easiest way of achieving networked live coding is perhaps through the use of `Tdefs`, which send messages to different servers. In figures 7.1 and 7.12–7.22, we could easily add a remote server to the event (e.g., `(note: x, dur: 0.125, server: x).play` or `Pdef(\x).set(\server, x)`). Furthermore, a `NodeProxy` or a `ProxySpace` can play just as well on a remote server as on a local one.[5]

Nevertheless, this is only half of what networked live coding is about: we are just as interested in the program text that gives rise to the sound as in the sound itself. An elegant way to share this source code is provided by the class `History`. Through adding a function to the `codeDump` variable of `Interpreter`, we are able to use `History` to distribute all evaluated code to all participants; furthermore, its GUI offers text filters for easy searching (see chapter 8), so new algorithmic sounds not only may become delocalized in the room, but their source code may be appropriated by others on the fly. (See figure 7.28.)

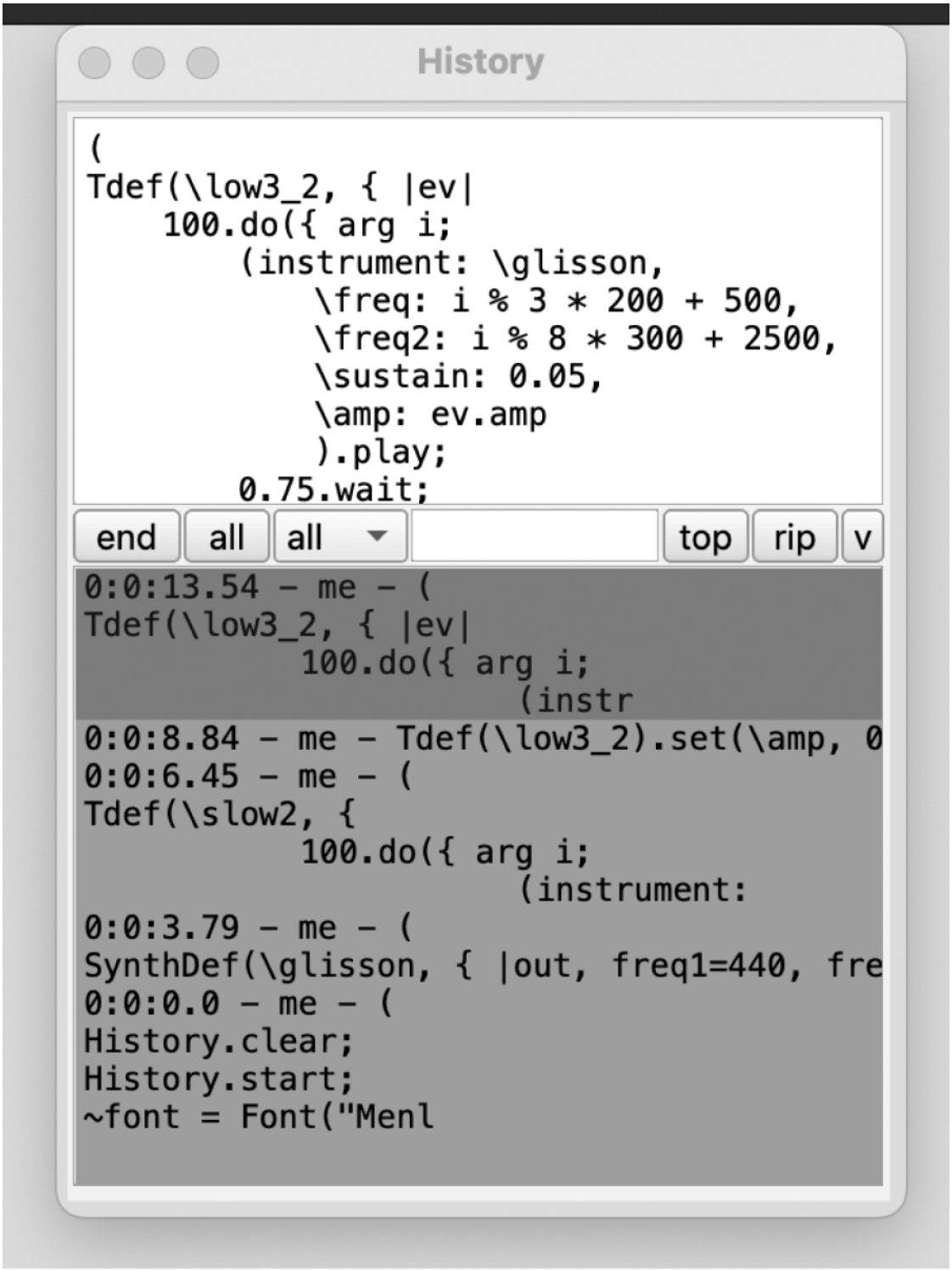


Figure 7.28
Distributed live coding with the History class.

For a more radical interlocking of performances, any `LazyEnvir`, such as `ProxySpace`, can be supplied with a dispatch function, which is called when objects are put in. As a result, an environment variable such as `~x` may be shared between participants, so that the modifications of some of the network affect a number of proxies

within the group. Where this is the case, the variable's identity becomes delocalized, and the refactoring that happens between the synth and the node graph (see section 7.3) is extended to different locations and persons. (The class `Public` in JITLibExtensions is such a dispatch.) As a result, one and the same variable may be used differently by each participant, so local changes result in a whole family of related, but not necessarily identical, changes in the entire ensemble (see Figure 7.29).

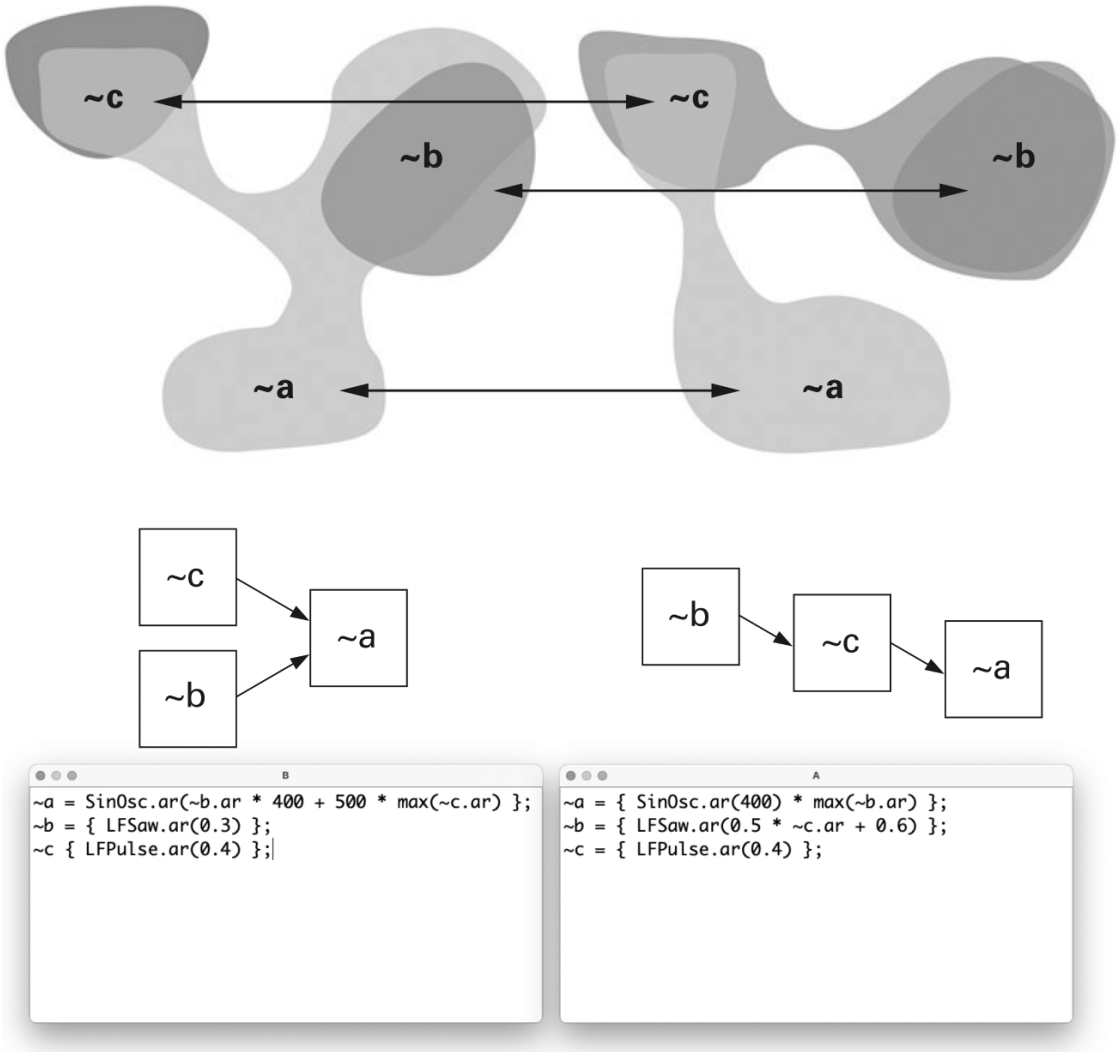


Figure 7.29
Shared variables between remote proxy spaces.

By modifying code at runtime, we include the programming activity in the flow of the program itself; changing our minds, we fold the representation into the process. This can become a fairly linear development in itself—never going back, always changing what is present as code. As a concluding example, available on the book website, eternal return shows how we can voyage back to past states. Here, past becomes future, as we

may edit past states that affect future placeholders. In time, we fold programming not only back into the program, but eventually into the memory of its alterations as well.

Notes

1. For more on live coding, see the references in this chapter and the papers listed at TOPLAP (http://toplap.org).
2. This procedure is typical for functional languages such as Haskell, in which all operations (such as currying) are essentially compositions of functions. In computer science and mathematics, such a function whose domain consists of other functions is sometimes called an "operator."
3. The actual order of nodes matters in cases where external audio input is processed. In such cases, one can reorder the nodes while playing `orderNodes(~x, ~y, ~z)`.
4. We leave as an open question whether it makes sense as a next step to map such streams that return symbols as an input to other streams that return symbols.
5. Note that one should not forget either to synchronize the system clocks of two computers or to set the server's latency to `nil`.

References

Clark, A., and D. Chalmers. 1998. "The Extended Mind." *Analysis*, 58(1): 10–23.

Collins, N., A. McLean, J. Rohrhuber, and A. Ward. 2003. "Live Coding Techniques for Laptop Performance." *Organised Sound*, 8(3): 321–330.

Guardans, R. 2010. "A Brief Note on the anwaa' Texts of the Late Tenth Century." In S. Zielinski and E. Fürlus, eds., *Variantology 4—On Deep Time Relations of Arts, Sciences and Technologies In the Arabic–Islamic World and Beyond*. Cologne, Germany: Verlag der Buchhandlung Walther König.

Iverson, K. 1980. "Notation as a Tool of Thought." The 1979 Turing Award Lecture. *Communications of the ACM*, 23(8): 444–465.

Klerer, M., and J. Reinfelds, eds. 1968. *Proceedings of the ACM SIGPLAN Symposium on Interactive Systems for Experimental Applied Mathematics*. New York: Academic Press.

Licklider, J. C. R. 1960. "Man–Computer Symbiosis." *IRE Transactions on Human Factors in Electronics*, 1: 4–11.

Mathews, M. V. 1969. *The Technology of Computer Music*. Cambridge, MA: MIT Press.

Matthews, H. F. 1968. "Venus: A Small Interactive Nonprocedural Language." In M. Klerer and J. Reinfelds, eds., *Proceedings of the ACM SIGPLAN Symposium on Interactive Systems for Experimental Applied Mathematics*, pp. 97–105. New York: Academic Press.

Rheinberger, H.-J. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford, CA: Stanford University Press.

Rohrhuber, J. 2008. "Network Music." In N. Collins and J. d'Escriván, eds., *The Cambridge Companion to Electronic Music*. Cambridge: Cambridge University Press.

Rohrhuber, J., and A. de Campo. 2004. "Waiting and Uncertainty in Computer Music Networks." In *Proceedings of the 2004 International Computer Music Conference*.

Rohrhuber, J., A. de Campo, and R. Wieser. 2005. "Algorithms Today—Notes on Language Design for Just-in-Time Programming." In *Proceedings of the 2005 International Computer Music Conference*, Barcelona, pp. 455–458.

Rohrhuber, J., A. de Campo, R. Wieser, J.-K. van Kampen, E. Ho, and H. Hölzl. 2007. "Purloined Letters and Distributed Persons." In *Proceedings of the Conference Music in the Global Village*. Available online at http://wertlos.org/articles/Purloined_letters.pdf.

Suchman, L. 2007. *Human–Machine Reconfigurations: Plans and Situated Actions*, 2nd ed. Cambridge: Cambridge University Press.

Vogt, K., W. Plessas, A. de Campo, C. Frauenberger, and G. Eckel. 2007. "Sonification of Spin Models: Listening to the Phase Transitions in the Ising and Potts Model." In *Proceedings of the 13th International Conference on*

Auditory Display. Montreal: Schulich School of Music, McGill University.

8 Object Modeling

Alberto de Campo, Julian Rohrhuber, and Till Bovermann

Programming can be thought of as expressing concepts in code for other human beings to understand. Especially in artistic work, but also in scientific contexts, concepts usually are initially "underspecified" if not "ill-defined," and one refines ideas iteratively. SuperCollider provides elegant ways to model objects by designing their behavior in a continuous flow while one is not yet completely sure of where the current exploration of ideas is leading.

In fully dynamic programming languages such as Smalltalk, a programmer may change a class while using the very same class for writing an e-mail or editing an image. In SuperCollider, this is not supported for runtime efficiency reasons; rather, classes and object models play different practical roles: class methods can be looked up very rapidly, and classes are very useful for capturing ideas that have mostly settled. Object modeling can be applied when working in a more dynamic live system is desired.

In this chapter, we provide some general background on object orientation from the perspective of object modeling (i.e., a way of simulating objects without creating full-fledged classes). We present examples written in this efficient and enjoyable development style and, discussing the flow of design decisions, we show how to convert object models to classes and methods if desired. We end with larger examples realized with object modeling.

8.1 Object Orientation, Behavior, and Polymorphism

The way in which a program works shows the labyrinth of consequences that follow the automatic, ignorant observance of rules implied by a concept. Thus, programming does not simply *express* preexisting concepts in code; often, concepts are also *formed* in the process of developing programs. To support such a coevolution of concepts and programs that embody them, programming languages have to provide very open and generic concepts. One of them is the *object*, or the paradigm of *object orientation*, which is a central idea in SuperCollider. Are objects the right metaphor for a programming language designed for sound synthesis research? At first glance, sound and objects may almost seem to be ontological opposites.

One of the original ideas of object orientation was a paradigm shift: whereas in procedural programming, the *subject* of a program (the operations/the operator) decided how to operate on the *object* of the program (the data), in object orientation, this is turned inside out. Here, it is the object which is responsible for the operations, and the closest thing to a subject are the other objects from which it receives messages in the call context. This shift made it possible to transpose the idea of a network of cooperating computers into a concept for programs themselves, in which every part is a miniature version of the whole; all that happens is that a large number of entities, composed of further entities, communicate with each other. Here, it becomes more obvious why objects may be a good metaphor for sound: it is all about conversation, exchange, resonance, and call and response.

Object orientation is often explained in terms of things that have properties and that belong to classes because they share properties: all cars, as a class, have wheels, and all bank accounts have a balance. If we know the class of an object, we know what messages it understands. But there is a different side to this, perhaps best described by Brian Cantwell Smith: "an object is something on which one can have a perspective" (1996, 117). Historically, the novelty of object orientation was not that it implements a taxonomy of things—computers have always been good for categorization anyhow. But it was a step in a new direction, insofar as data and operations were seen as inseparably linked. In his description of how this paradigm emerged in the late 1960s, Alan Kay noted how indexing into an array (the "at" operation in `a = [1, 0, 0, 1, 1, 2, 0, 1]; a.at(5);`) was better *not* seen as an operation (*at*) on data (*the array*); rather, it was the array that *behaved* in a certain way in response to the message "*at*." In 1993, Kay wrote: "It took me a remarkably long time to see this, partly I think because one has to invert the traditional notion of operators and functions, etc., to see that objects need to privately own all of their behaviors: *that objects are a kind of mapping whose values are its behaviors*" (1993, 8). As a consequence, we are not confined to thinking of an object only in terms of its descent, its parent classes; we also can think of it purely in terms of what it responds to. The fact that different objects may respond to the same message differently is called *polymorphism*, and, as we will see, that makes it easy to extend and modify a system. Only behavior matters; this notion is sometimes called "duck typing": *If it looks like a duck, swims like a duck, and quacks like a duck, then it's probably a duck.*

Looking at what kinds of objects respond to a given message in SuperCollider, we find many examples that do not share any common class (apart from the mother of all objects, the class `Object`). The most heterogeneous entities may respond to the same message. Take, for example, the message squared. Any object that responds to it can be used in an expression such as `x.squared`. So if all that really matters about an object is how it behaves, then the same expression can be used in many different contexts without

changing anything: we may square integers and floats, and also unit generators, streams, and patterns. We can square not only arrays of integers, but also arrays of arrays of integers and arrays of arrays of unit generators. This system is extendable for new concepts with minimal effort: by defining a new kind of object that responds to `squared`, we may square not only that object, but also arrays of objects.

The statement "everything is an object" can be complemented with "everything behaves." This is true even when an object does *not* understand a given message; how it then responds is specified in its `doesNotUnderstand` method definition. Normally, this is useful for providing error information (see, e.g., the error created by `{SinOsc.ar}.pay`). This also means that if we want to define new behavior of an object dynamically without recompiling the class library, we can override `doesNotUnderstand` in a subclass to forward the selector (the message name that was not understood) and the argument (*args*) to any functionality we like. As discussed in chapter 5, the class `IdentityDictionary`, as well as its subclasses `Environment` and `Event`, do this: when its instance variable `know` is `true` (so it always "knows"), the dictionary will check whether it contains a value defined for the key that was the message name that it did not understand. Consequently, object prototyping can be done without losing the old state of the system; one simply adds or removes a function in the Event (or an instance of an equivalent class) and, by doing so, changes its behavior. Such an object can be passed around in the system and used for calculation, just like any other object. This is how object modeling begins.

8.2 Common Compromises in Programming Languages

Every programming language is based on compromises, and there is no advantage without a disadvantage; languages are designed with different tasks in mind, so an idea can be very simple to realize in one language and very tricky in another. In computer science, these design aims are very thoroughly (and sometimes heatedly) discussed.

Some design aims for programming languages are *conceptual purity*, which many functional languages aim for; *elegance*, which can be rather elusive (see, e.g., Chaitin, 1998); *"naturalness" of use*—among other things, Smalltalk was designed for children to learn programming easily; *extensibility* (i.e., being able to add features to the language as needed); *runtime efficiency*, which many lower-level languages such as C are designed for; *programming time efficiency*, being able to realize complex systems within reasonable amounts of time; *scalability*, being able to extend architectures up to very big and complex systems; *real-time performance*, which is critical for music systems for live use; *generality*, covering a wide range of domains; *domain-specific features*, such as doing audio DSP efficiently; and (maybe most important) *support of different programming styles*.

Existing languages influence programming styles, just as programming styles influence new language design. SuperCollider has absorbed and adapted many concepts and styles from other languages, such as object orientation from Smalltalk, Ruby, and Dylan; syntax from C++; functional notation from Lisp; concepts for multidimensional arrays from the J language; and list comprehensions from functional languages like Haskell. Some details are to be found in the Reference and Overview pages in the documentation.

Naturally, no language can achieve all these objectives fully, so every language essentially rests on a set of trade-offs. By comparison, SuperCollider tends to be oriented toward efficient real-time performance, flexibility to express concepts in very many different ways, and programming time efficiency. This is why it makes sense to keep a distinction between a static class library, on the one hand, and dynamic object modeling, on the other.

8.3 Keeping Things Around

To be able to work fluently on groups of objects, they must persist long enough. The simplest option is to keep objects as interpreter variables (which are global for runtime code), using the lowercase letters `a–z`. However, there are only 25 letters (`s` is not counted because by convention, it is reserved for the server), so one runs out of names quickly, and it can be hard to remember which letter was which object.

A second option is keeping things in the current environment, as in `~melA = [0, 2, 3]`, which allows more descriptive names. For larger projects, which may use the same terminology in different contexts, it is still inconvenient that this is a single, flat space of names. Furthermore, for using special environments such as `ProxySpace` (see chapter 7), the `currentEnvironment` is often replaced; switching between different environments can be confusing.

A third option is keeping environments (or events, for syntactic brevity) as global variables and storing things in them by name:

```
q = (); // assign a new Event to the interpreter variable q
q[\melA] = [0, 2, 3]; // store an array of values in the event
```

Equivalently, `put` and `at` operations can be written like `getter` and `setter` messages for what could be called “pseudoinstance variables”:

```
q.melA_([0, 2, 3]);
```

or as

```
q.melA = [0, 2, 3];
q.melA + 7;
```

One can also organize repositories of objects hierarchically:

```
q.melodies = ();
q.melodies.melA = [0, 2, 3];
```

The messagelike syntax allows a different perspective: we can say that the event `q`, taken as an object, has acquired new behavior with new message names. Since by design the object `Event` does not understand a method like `melA` or `melodies`, these method calls are redirected. In the case of `setter`-like methods like `melA_`, the event's `put` method is evoked, which stores the object `[0, 2, 3]`, under the key `'melodies'`. Calling `q.melodies` does the inverse; since the method is not understood, it is redirected to look up the object that the event currently contains at the key `'melodies'` and return it. Why this is a way to implement a behavior of the event object becomes clearer when one considers how functions are treated as so-called pseudomethod calls: when the name is called, the function is evaluated, taking the event itself as the first argument followed by all other arguments:

```
q.playMel = {|ev, transpose = 0| Pbind(\note, Pseq(ev.melA) +   tra
nspose, \dur, 0.2).play};
q.playMel(0);
q.playMel(3);
```

8.4 Classes and Events as Object Models

SuperCollider allows very efficient implementations of objects and their behavior as classes. Figure 8.1 shows an example of a simple object called `Puppet`, written as a class, and some tests of its methods.

```
Puppet {
   var <>myfreq;   // an instance variable with a getter and a sett
er method
        // a method for creating a new object of this kind
   *new {|myfreq=50| ^super.new.myfreq_(myfreq)}

        // a simple method that uses 'myfreq' for something audibl
e.
   blip {{Blip.ar(myfreq, 11) * XLine.kr(1, 0.01, 0.6,   doneActio
n: 2)} .play;}
```

```
}

   // tests for the behavior implemented so far:
m = Puppet.new(50); // make an instance of Puppet, pass in myfreq

m.dump;             // test that myfreq is set correctly
m.myfreq;  // test accessing myfreq
m.blip;       // should sound
m.myfreq_(100);  // test setting myfreq
m.blip;       // each should sound differently
```

Figure 8.1
A `Puppet` class, and some tests for it.

When working with objects as classes, making a single change or addition to an object's behavior requires (1) changing the source file (e.g., *Puppet.sc*) and saving it, (2) recompiling the class library, and (3) rebuilding the state of things needed for testing the new behavior. In other words, we need to make and maintain a script to get back to where we were in order to continue working; putting this into the startup file is convenient. Here, the rebuilding script is simply `m = Puppet.new(50);` but often, it is much longer.

The same design steps can be taken with an `Event` to model the new object (see figure 8.2).

```
m = ();          // make an empty event
m.myfreq_(50);   // put something in it with a setter method:
                 // a pseudo-instance variable
m.myfreq;        // look it up with a getter method
                 // put a function into it with a setter:
                 // this becomes a pseudo-method
m.blip_({|ev| {Blip.ar(ev.myfreq, 11) * XLine.kr(1, 0.01, 0.6,   do
neAction: 2)} .play;});
m.blip; // execute the function with a pseudo-method call   (same n
ame)
```

Figure 8.2
A puppet modeled as an event

Now, changing or adding more variables and methods to `m` is extremely simple (see figure 8.3).

```
(
m.numHarms_(20);     // a new instvar
```

```
m.decay_(0.3);  // and another
   // update the blip method to use them:
m.blip_({|ev|
   {Blip.ar(ev.myfreq, ev.numHarms)
   * XLine.kr(1, 0.01, ev.decay, doneAction: 2)} .play;
});
)
m.blip; // test
```

Figure 8.3
Add more instance variables and change the blip method.

Note the use of the argument `| ev |` of the function for the pseudomethod: this makes the event itself (the modeled object) accessible inside the pseudomethod in the way that an object is accessible inside one of its methods through the special variable `this`.

There are two important caveats. When a pseudomethod name that already exists is used as a regular method name, the event (i.e., the object model) will call the "real" method, not the pseudomethod. Since this can result in quite obscure bugs, a warning is posted when we try to create a pseudomethod with such a conflicting name (e.g., try executing `m.size_(12)`). Accessing the variable with `m.size` returns 5 (if the event `m` currently contains five objects), whereas `m[\size]` returns 12, the value previously stored. This can be remarkably confusing.

A second potential problem is that misspelled messages do not throw errors: after having set `q.puppet = ()`, the message `q.pupet` (spelled incorrectly) does not complain, as a misspelled real method would; it just returns `nil`.

The rest of this chapter goes through larger examples in ascending complexity; whereas in the first example, every design decision is discussed thoroughly, the later examples are viewed more generally, considering issues from a longer distance. The focus is on the working style and the flow of decisions; both could be shown equally well with many other examples.

8.5 Case Study 1: A Shout Window

8.5.1 Background

Two of the authors used to perform in the ensemble Powerbooks UnPlugged,[1] a band where both code and sound are shared among all players; whenever one player executes a codelet, this small piece of code is sent to all others by network and becomes available in the `History` window. (`History` keeps a log of all the code executions and can display them for later access, such as for rewriting; see also the `History` Help file.)

In performance, discussing flow and structural decisions (such as “move to next section?” and “come down to end?”) goes through the same mechanism, which causes problems: while one is reading or rewriting code, it is easy to miss calls for discussion or attention messages. Because the band members sit distributed within the audience, eye contact is unreliable.

Thus, we needed a simple-to-use and minimal mechanism for writing messages to all the players that are displayed very prominently. Tom Hall’s SuperCollider extension honk, which uses the freeware tool BigHonkingText, provided a clear idea of how to display messages so they are very difficult to miss. After seeing them displayed in this fashion (as a large, top-level, floating window), Alberto de Campo and Hannes Hölzl wrote a similar utility in SuperCollider in a joint session, using the object-modeling approach described; here, we go through a reconstruction of the design steps, showing the concepts at work.

8.5.2 An Object Model

First, we make an empty event and create two variables: a window and a text view. Then we tune their appearance (see figure 8.4).

```
z = z? ();  // make an empty event as a pseudo-object
z.window = Window("Shout", Rect(0, 0, 1200, 80)).front;
z.textView = TextView(z.window, z.window.bounds.extent);
z.textView.resize_(5);
z.textView.font_(Font("Monaco", 64));
z.textView.string_("Shout this!");

// tune appearances
z.window.alwaysOnTop_(true);  // make sure it is always on top.
z.window.alpha_(0.5);    // make the window semi-transparent

z.window.close;  // close when done
```

Figure 8.4
A minimal shout window sketch.

When the model is good enough, we can pull it into a single pseudomethod and test it (see figure 8.5).

```
(
z.makeWindow = {|z, message="Shout this!"|
   z.window = Window("Shout", Rect(0, 0, 1200, 80)).front;
   z.window.alwaysOnTop_(true);
```

```
    z.window.alpha_(0.5);

    z.textView = TextView(z.window, z.window.bounds.extent);
    z.textView.resize_(5);
    z.textView.font_(Font("Monaco", 64));

    z.textView.string_(message);
};
)
z.makeWindow;
z.makeWindow("Try showing that.");
```

Figure 8.5
Wrap the sketch in a pseudomethod

Next, we add more methods to our object model and test them. Calling `setMessage` sets only the message displayed, whereas `shout` is meant to be the top-level interface. Create a window, if needed, and display the message (see figure 8.6).

```
z.setMessage = {|z, str| z.textView.string_(str)};
z.setMessage("Does this update?");  // test
(
// use z.shout as main interface
z.shout = {|z, str|
    if (z.window.isNil or: {z.window.isClosed}) {
        z.makeWindow(str)
    }{
        z.setMessage(str)
    }

};
)
z.shout("Do we get this?"); // test

z.window.close;
z.shout("Do we get this too?"); // also when window has closed?
```

Figure 8.6
More pseudomethods.

When only the text changes on an existing window, that still might be ignored under performance conditions. So we flash the text in several colors (figure 8.7).

```
z.textView.stringColor_(Color.red);  // try a single color
(
z.animate = {|z, dt=0.2, n = 6|
   var colors = [Color.red, Color.green, Color.black];
   fork {
        n.do {|i|
              dt.wait;
              defer {z.textView.stringColor_(colors.wrapAt(i))}
        }
   };
};
)
z.animate;           // test with default values
z.animate(0.1, 24); // and test with arguments given
```

Figure 8.7
Text color animation.

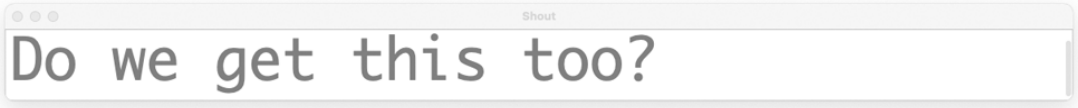


Figure 8.8
A screenshot of the shout window so far.

How best to use this in performance? Typing something like `z.shout("OK, take it to the bridge!")` seemed rather clumsy; in discussion, we came up with writing `Shout` messages as comments with a special shout prefix for comment lines:

```
//!! this is a comment line with a 'shout tag' prefix.
```

Executing this line of comment does nothing, but it should be recognized as a message to be shouted, and forwarded to the `Shout` window automatically. The code-forwarding mechanism in the networking setup for Powerbooks UnPlugged uses the interpreter variable `codeDump`. By default, `codeDump` is `nil`; putting a function here allows additional reactions to the string after it has been interpreted (see figure 8.9).

```
this.codeDump = {|str, result, func| [str, result, func]   .printAl
l};

a = 1 + 2;  // code appears in post window now
```

```
z.shoutTag = "//!!";
this.codeDump = {|str|
if (str.beginsWith(z.shoutTag)) {z.shout(str.drop(z.shoutTag   .siz
e))}};
//!! a comment with a 'shout tag' now gets shouted!
```

Figure 8.9
Using `codeDump` to shout.

Playing around with `shout` messages of different lengths revealed that the font size should adjust to the message such that the message is displayed as large as possible, but always in full length. Intuitively, longer messages should use smaller fonts, so roughly, `fontsize = constant * viewWidth/messageSize`. Quick tests led to a constant of 1.64 for the Monaco font, which has fixed character spacing. (Although line wrapping of the text view would be a nice option, we decided not to spend time on that.) To use font resizing in `z.makeWindow` as well, we rewrite it to use `z.setMessage` (see figure 8.10).

```
(
z.setMessage = {|z, str|
   var messSize = str.size;
   var fontsize = (1.64 * z.textView.bounds.width)   / max(messSiz
e, 32);
   z.textView.font_(Font("Monaco", fontsize));
   z.textView.string_(str);
   z.animate;
};
)
//!! a long comment gets scaled down to a rather smaller font size,
minimally fontsize 32!
//!! a short message is big!
```

Figure 8.10
Updated `setMessage` to scale font and flash text.

This object model fully fits the purpose as is, and all customization (such as preferred window position, font and size, animation colors, rate, and duration) can easily be performed by editing those details. The best way to maintain such an object model setup is to put it in a separate file that can be run at once, with documentation notes, commented-out tests, and examples for intended uses. The book code repository has an example file for the Shout setup.

8.5.3 Conversion to a Class

Keeping concepts in object-modeling implementation permanently has its benefits: the flexibility of easy experimental adaptation, and no need to recompile after changes benefits of classes. But sometimes a design converges sufficiently that it becomes preferable to turn it into a stable class. Generally, this conversion is simple: all pseudovariables become class or instance variables, and all pseudomethods become class or instance methods.

In our specific case, we only ever want a single `Shout` window, and thus we only need a single `Shout` object, so we can move all variables and methods in the class itself, and in fact forgo making any instances of the class. (This is a special form of the `Singleton` pattern.) Thus, `z.window`, `z.textView`, and `z.shoutTag` become class variables `window`, `textView`, `tag = "//!!"`. Note that `window` and `textView` can be accessed but not set, whereas `tag` can be changed because it is written with accessor shortcuts.

The following code is also available in the book code repository as a folder with consecutive evolving versions of the `Shout` files called *Shout11.scd*, *Shout12.scd*, etc. To try them out, we can place this folder in SuperCollider's user extensions folder at `Platform.userExtensionDir`, and enable the class version to test by renaming it `Shout11.sc` (See figure 8.11.)

```
// beginning of file—Shout11.scd
Shout {
    classvar <>tag="//!!";
    classvar <window;
    classvar <textView;
}
// end of file—Shout.sc
```

Figure 8.11
A Shout class.

Added here are the class variables `rect`, a rectangle where to place the window, and `codeDumpFunc`, the redirecting/sending function. By default, `rect` assumes full window width and placement at the bottom, but this can be adjusted as desired. The `codeDumpFunc` is initialized in `*initClass`, which is always called upon compilation. (See figure 8.12.)

```
/* tests
Shout.rect;
Shout.codeDumpFunc;
```

```
*/
Shout {
    classvar <>tag="//!!";
    classvar <rect;
    classvar <window;
    classvar <textView;
    classvar <codeDumpFunc;

    *initClass {
        rect = Rect(0, 0, Window.screenBounds.width, 80);
        codeDumpFunc = {|str|
            if (str.beginsWith(tag)) {
                Shout(str.drop(tag.size))
            }
        };
    }
}
```

Figure 8.12
More class variables and an `initClass` method.

Next, we turn `z.makeWindow` into a class method in `Shout`, using the `rect` variable for window size and placement. As `z.makeWindow` calls `z.setMessage`, we also add a simple first `setMessage` method (see figure 8.13).

```
Shout {
    . . .
        *makeWindow {|message="Shout this!"|
            window = Window("Shout", rect).front;
            window.alwaysOnTop_(true);
            window.alpha_(0.5);

            textView = TextView(window, rect.extent);
            textView.resize_(5);
            textView.font_(Font("Monaco", 64));
            this.setMessage(message);
    }
    *setMessage {|message|
        textView.string = message;
    }
    . . .
}
```

```
// tests:
Shout.makeWindow("Blong");
```

Figure 8.13
Adding `*makeWindow` and `*setMessage` to `Shout`.

Next is `z.shout`, which was the top interface method to use in the object model. Transferring it to a `Shout.shout` method seems clumsy, but repurposing the `*new` method is interesting here: `Shout.new("message")` can also be written as `Shout("message");` for symmetry, we also add a `*close` method that tries to close the window (see figure 8.14).

```
z.shout = {|z, str|
    if (z.window.isNil or: {z.window.isClosed}){
        z.makeWindow(str)
    }{
        z.setMessage(str)
    }
};

Shout {
    . . .
    *new {|message = "Shout!"|
        if (window.isNil or: {window.isClosed}){
            this.makeWindow(message);
        }{
            this.setMessage(message)
        }
    }
    *close {
        if (window.notNil and: {window.isClosed.not})
         {window.close}
    }
    . . .
}
// tests:
Shout("Test 1, 2");
Shout("Test 1, 2, 3, 4");    // same window
Shout.close;
Shout("Test 1, 2");       // new window
```

Figure 8.14
Converting `z.shout` to `Shout.new`.

If we want to use `"message".shout`, we can add an extension method `shout` to the `String` class very simply (see *extStringShout.sc*):

```
+ String {
     shout {Shout(this)}
}
```

Also, `z.animate` transfers easily (see figure 8.15). Note that we make `colors` a class variable which can be customized, and if not set, it gets initialized lazily the first time that `*animate` is used.

```
z.animate = {|z, dt=0.2, n = 6|
   var colors = [Color.red, Color.green, Color.black];
   fork {
        n.do {|i|
              dt.wait;
              defer {z.textView.stringColor_(colors.wrapAt(i))}
        }
   };
};

Shout {
   . . .
   classvar <>colors;
   . . .
   *animate {|dt=0.2, n=6|
        colors = colors?? {[Color.red, Color.green, Color.black]};
        fork {
              n.do {|i|
                   dt.wait;
                   defer {textView.stringColor_(colors.wrapAt(i))}
              }
        };
   }
   . . .
}
// tests:
Shout("Test 1, 2");
Shout.animate;
```

Figure 8.15
Converting `animate` to a class method.

Now `setMessage` can be refined with font scaling and calling `animate`. (See figure 8.16.)

```
z.setMessage = {|z, str|
    var messSize = str.size;
    var fontsize = (1.64 * z.textView.bounds.width) /   max(messSiz
e, 32);
    z.textView.font_(Font("Monaco", fontsize));
    z.textView.string_(str);
    z.animate;
};

Shout {
    . . .
    *setMessage {|message|
         var messSize, fontSize;
         messSize = message.size;
         fontSize = (1.64 * width) / max(messSize, 32);

    defer {
         textView.font_(Font("Monaco", fontSize))
         .string_(message.asString);
    };
    this.animate;
    }
    . . .
}
// tests:
Shout("Test 1, 2");
Shout("Test" + (1..16));
```

Figure 8.16
Converting `setMessage`.

Some usability goals came up at this point: it would be nice to turn shouting on and off easily; and when a shout comes in, the current text document should stay in front so that one can continue typing. The first step is turning `Shout` on and off; the safe and polite way to use `codeDump` is to assume that others may have added functions to it, and we can do this easily with `addFunc`/`removeFunc` (see FunctionList.help). First, we try this in the object model, then we add class methods. (See figure 8.17.)

```
// load object model z first. . .then
// clear codeDump
```

```
this.codeDump = nil;
// declare the func to add/remove
z[\codeDumpFunc] = {|str|
   if (str.beginsWith(z.shoutTag)){
        z.shout(str.drop(z.shoutTag.size))
   }
};
z.enable = {
   this.codeDump = this.codeDump.addFunc(z[\codeDumpFunc]);
};
z.disable = {
   this.codeDump = this.codeDump.removeFunc(z[\codeDumpFunc]);
};
// tests:
z.enable
//!! should shout
this.codeDump.postcs   // should be there now
z.disable
//!! shhhhh // should be silent
this.codeDump.postcs   // should be gone now

Shout {
   . . .
   *enable {
        var interp = thisProcess.interpreter;
        this.disable; // remove codeDumpFunc first
        interp.codeDump = interp.codeDump
        .addFunc(codeDumpFunc);
   }
   *disable {
        var interp = thisProcess.interpreter;
        interp.codeDump = interp.codeDump.removeFunc   (codeDumpFun
c);
   }
   . . .
}
// tests
Shout.add;
//!! test whether Shout works now—it should!
Shout.remove;
//!! test whether Shout works now—should be off.
```

Figure 8.17
Handling `codeDump` in object model and class.

For distributing messages to networked performers, one could modify `codeDumpFunc` to broadcast `Shout` messages, for instance, by sending messages to the local network's broadcast address, with an `OSCFunc` listening for them and doing the shouting when a message comes in. Such an example would go into a Help file for `Shout`.

Versions of `Shout` were included in the (now outdated) `Republic` quark, and later as `NMLShout` in the `Utopia` quark; we continue to use them in network concerts, now often with the `HyperDisCo` quark, and they have facilitated communication within the ensembles considerably.

8.6 Example 2: Aspects in QCD Sonification

The techniques described above are often sufficient for rapid prototyping of classes in SC3, but sometimes the development process of a desired application is more complex. Although some parts settle sufficiently to warrant making them a class, others need to remain open for longer experimentation, especially when implementing both data preparation (complex routines for getting and preprocessing data) and complex data usage (which must be flexible for experiments). For the scientific context shown here, it was desirable to keep some options open even longer; we wanted to allow users to explore further ideas for data usage, allowing them to test new hypotheses at runtime.

The quantum chromodynamics (QCD) sonification environment was developed within the SonEnvir project in 2006, with Katharina Vogt as the main physics researcher, Till Bovermann as the SC3 specialist, and Philipp Huber preparing QCD lattice data.[2] We created a system that both uses classes and has the possibility of modifying the functionality of its parts at runtime.

QCD is the theory of the strong interaction between quarks and gluons. Partner researchers of the SonEnvir project computed several kinds of QCD model data. Each data item is an element of a high-dimensional vector space ($R^{16 \times 16 \times 16 \times 32}$) and was provided to us split into 4 csv text files of 2.6 MB each. Each file holds either the raw data or 1 of 3 so-called *smearing steps*, as precomputed by our colleagues.

Reading in the data text files (with `CSVFileReader`) allowed the first audible experiments by serializing the numerical values and playing the resulting time series like an audification (see chapter 13). But reading in all four files of one data item took quite long, so we wrote a converter from `csv` to `aiff` files; this was fast enough for interactive comparison between different data items and smearing steps.

Discussion revealed that the audification used in the first pass produced artifacts due to the arbitrary way the high-dimensional data were put into a sequence. We concluded that for experimentation, we required runtime choice between sequencing algorithms

and runtime selection of a region of interest in the data item under exploration. One should be able to navigate a region through the data set, selecting its position and radius by mouse over, mouse click, or keyboard events, or by hardware interfaces. Furthermore, both sequencing algorithm and sonification strategy should be modifiable at runtime.

While prototyping these features we found that half the code for an example was for data preparation and visualization setup. Therefore, we moved all the data preparation and setup into a class while providing code interfaces to user-adjustable functionality, leaving possibilities to add and modify them at runtime. For each user-changeable algorithm, we created a dictionary in the QCD class, filled with at least one trivial algorithm written as a function with a fixed interface. Thus, instead of calling a fixed sequencing algorithm, such as `flat` (see figure 8.18), we called the function in a dictionary at a specific key (see figure 8.19).

```
serialize {|index, pos, extent = 8|
    var ranges, slice;
    // get ranges
    ranges = pos.collect{|pos, i|
        ((pos-(extent*0.5))..(pos+(extent*0.5)-1)) % this.shape[i]
    };
    // get sub-slice
    slice = this.slice(index, *ranges);
    // trivial serialization of multidim. slice
    ^slice.flat
}
```

Figure 8.18
A fixed serialization method.

```
serialize {|index, pos, extent = 8, how = \hilbert|
    // [. . .]
    slice = this.slice(index, *ranges);
    // call function in serTypes dictionary
    ^serTypes[how].(slice.asArray, extent)
}
```

Figure 8.19
Flexible serialization by lookup.

Now, one can provide several methods to choose from within `*initClass` and add more choices later, while experimenting. (See figures 8.20 and 8.21.)

```
*initClass {|numDims = 4|
    . . .
    // slice here is a 4d hypercube of extent <extent>
    serTypes = (
         hilbert: {|slice, extent|
               extent.isPowerOfTwo.not.if({
                   "QCD:serialize: extent has to be a power of   tw
o".error
               });
               HilbertIndices.serialize(slice)
         },
         torus: {|slice, extent|
               slice.flat;
         },
         scramble: {|slice|
               slice.flat.scramble;
         }
    );
}
```

Figure 8.20
Some initial serialization methods.

```
QCD.serTypes.put(\star, {|slice|
    var starSize = slice.size div:2;
    var numDims = 4;
    var starShape;

    starShape = neighbours1.collect({|nb|
         (0..starSize).collect(_ * nb)
    }).flatten(1).collect {|indexN|
         indexN + (starSize.div(2)+1).dup(numDims)
    };

    starShape.collect{|iA| slice.slice(*iA)}
});
```

Figure 8.21
Adding a new serialization type at runtime.

This mixed approach of classes and the object-modeling technique combines the efficiency of classes (also in hiding settled functionality) with the flexibility to try new ideas at runtime—in the case of the serialization by adding functions to the `serTypes`

dictionary. Applying the same strategy for sonification variants proved helpful in understanding which representations seemed more expressive of data properties.

8.7 Example 3: A Miniature CloudGenerator

As the final example, we demonstrate a slightly larger project written in object modeling and just-in-time style (see chapter 7). CloudGenMini is based on a classic granular synthesis program by Curtis Roads and John Alexander, CloudGenerator, which creates clouds of sound particles based on statistical distributions. While the present discussion focuses on coding style, the synthesis technique and aesthetic aspects are discussed in chapter 16.

`CloudGenMini` combines several components: a selection of SynthDefs used to generate single granular sounds, a task that generates a cloud of sound particles, functions that provide random ranges for the control parameters and to store them, functions to cross-fade between stored settings, and, finally, a lightweight GUI for playing the instrument. CloudGenMini can be run entirely from the file Ch8_ex4_CloudGenMini.scd (in the code repository).

Figure 8.22 shows two of the SynthDefs. It is recommended practice to write tests which verify that all parameters in a SynthDef work as expected. We can also learn to find good parameter ranges in this way.

```
(
   // a gabor (approx. gaussian-shaped) grain
SynthDef(\gab1st, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = SinOsc.ar(freq);
   var env = EnvGen.ar(Env.sine(sustain, amp * AmpComp.ir(freq)   *
0.5), doneAction: 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;

               // a simple percussive envelope
SynthDef(\percSin, {|out, amp=0.1, freq=440, sustain=0.01, pan|
        var snd = FSinOsc.ar(freq);
        var env = EnvGen.ar(
        Env.perc(0.1, 0.9, amp * AmpComp.ir(freq) * 0.5),
              timeScale: sustain, doneAction: 2
        );
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;
/*
   // tests for the synthdefs:
```

```
Synth(\gab1st);
Synth(\gab1st, [\out, 0, \amp, 0.2, \freq, 2000, \sustain, 0.05]);
Synth(\gab1st, [\out, 0, \amp, 0.2, \freq, 20, \sustain, 0.05]);
Synth(\percSin);
Synth(\percSin, [\amp, 0.2, \sustain, 0.1]);
*/
);
```

Figure 8.22
Some granular SynthDefs and tests.

Figure 8.23 creates an event `q` and provides globally accessible names for parameter ranges, as well as specs for grain parameters. A named `TaskProxy` is created with `Tdef(\cloud0)` (see the `Tdef` Help file), and its internal environment is initialized with a default SynthDef name to use; also created is an event called `current`, which contains default settings for all the parameter ranges. This design already makes the necessary preparations for using multiple settings (i.e., presets).

```
(
q = q? ();

   // some globals
q.paramRNames = [\freqRange, \durRange, \densRange, \ampRange,   \p
anRange];
q.paramNames = [\freq, \grDur, \dens, \amp, \pan];
q.syndefNames = [\gab1st, \gabWide, \percSin, \percSinRev,   \percN
oise];

   // specs for some parameters
Spec.add(\xfadeTime, [0.001, 1000, \exp]);
Spec.add(\ring, [0.03, 30, \exp]);
Spec.add(\grDur, [0.0001, 1, \exp]);
Spec.add(\dens, [1, 1000, \exp]);

   // make an empty tdef that plays it,
   // and put the cloud parameter ranges in the tdef's // environme
nt
Tdef(\cloud0)
   .set(
   \synName, \gab1st,
   \vol, 0.25,
   \current, (
        freqRange: [200, 2000],
```

```
        ampRange: [0.1, 1],
        durRange: [0.001, 0.01],
        densRange: [1, 1000],
        panRange: [-1.0, 1.0]
    )
);

        // make the tdef that plays the cloud of sound // particles
here,
        // based on parameter range settings.
Tdef(\cloud0, {|e|
    loop {
        s.sendBundle(s.latency, [
                "/s_new", e.synName? \gab1st,
                -1, 0, 0,
                \freq,   exprand(e.current.freqRange[0], e.current.
freqRange[1]),
                \amp,   exprand(e.current.ampRange[0], e.current.   a
mpRange[1]) * e.vol,
                \sustain,   exprand(e.current.durRange[0], e.current.
durRange[1]),
                \pan,   rrand(e.current.panRange[0], e.current.   pan
Range[1])
                ]);
                exprand(e.current.densRange[0].reciprocal, e.current.
densRange[1].reciprocal).wait;
    }
}).quant_(0);
)
```

Figure 8.23
Global setup and a player `Tdef` for the cloud.

The `Tdef` function itself is very simple: in a loop, it generates random values for the next grain within the ranges for each parameter stored in the current settings. Using `s.sendBundle` is low-level but very efficient messaging, which is useful for high-density granular clouds. The provided tests exercise all the behaviors defined so far: playing and stopping the cloud; putting into the current settings new ranges for `densRange`, `freqRange`, and so on; verifying that they change the running cloud; and changing the grain SynthDef used. (See figure 8.24.)

```
Tdef(\cloud0).play;
```

```
    // try changing various things from outside the loop.
    // change its playing settings
Tdef(\cloud0).envir.current.put('densRange', [50, 200]);   // dens
e, async
Tdef(\cloud0).envir.current.put('densRange', [1, 10]);   // sparse,
async
Tdef(\cloud0).envir.current.put('densRange', [30, 30]);   // synchr
onous
    // for faster access, call the tdef's envir d
d = Tdef(\cloud0).envir;
d.current.put('freqRange', [800, 1200]);
d.current.put('durRange', [0.02, 0.02]);
d.current.put('ampRange', [0.1, 0.1]);
d.current.put('panRange', [1.0, 1.0]);
d.current.put('panRange', [-1.0, 1.0]);
d.current.put('densRange', [30, 60]);
d.synName = \percSin;
d.synName = \gab1st;
d.synName = \gabWide;
d.synName = \percSinRev;
d.synName = \percNoise;
d.synName = \percSinRev;
d.synName = \gab1st;
d.current.put('durRange', [0.001, 0.08]);
```

Figure 8.24
Tests for the cloud.

A common heuristic for exploring the possibilities of a synthesis process is to employ random ranges, which may reveal more interesting areas than one might encounter by making changes manually. Figure 8.25 shows a method for specifying random ranges for all parameters based on the global maximum settings defined in figure 8.23. This method is then employed for creating eight random settings, so one can try switching among different cloud parameter states. The concept of presets and interpolation goes back to SC2 classes written by Ron Kuivila. He noted that he was inspired to implement this multidimensional interpolation by David Tudor, who would play a mixer polyphonically with five fingers at a time. This approach is available in the `Conductor` quark. Many of the ideas explored here were later expanded on in the `ProxyPreset` class and its subclasses for the various JITLib proxies, available in the `JITLibExtensions` quark.

```
(
   // make the Tdef's envir a global variable for easier // experim
enting
d = Tdef(\cloud0).envir;
   // a pseudo-method to make random settings, kept in the // Tde
f's environment
        // randomize could also do limited variation on existing //
setting.
d.randSet = {|d|
   var randSet = ();
   q.paramRNames.do {|pName, i|
        randSet.put(pName,
              q.paramNames[i].asSpec.map([1.0.rand, 1.0.rand].sort)
        );
   };
   randSet;

};
/*    test randSet:
d.current = d.randSet;
*/

// make 8 sets of parameter range settings:
d.setNames = (1..8).collect {|i| ("set" ++ i).asSymbol};
d.setNames.do {|key| d[key] = d.randSet;}

/*    test switching to the random presets
d.current = d.set1.copy;    // copy to avoid writing into a   // st
ored setting when it is current.
d.current = d.set3.copy;
d.current = d.set8.copy;
*/
);
```

Figure 8.25
Making random settings and eight random presets to switch from one to the other.

The original CloudGenerator program creates clouds by specification: one sets values for cloud duration, grain duration, density, and amplitude, and for high and low band limits (i.e., minimum and maximum grain frequencies); setting start and end values allows one to define a *tendency mask* for grain frequency. In `CloudGenMini`, this is generalized to setting ranges for all parameters and being able to cross-fade between

range settings for every parameter. (To make a parameter deterministic, one simply sets its minimum and maximum to the same value.) The new functions in figure 8.26 allow us to specify defined-length clouds based on tendency masks: a flag `d.stopAfterFade` is used for the optional ending when a parameter cross-fade has ended; `d.xfadeTime` sets cross-fade time. A `TaskProxy` (see chapter 7), `d.morphtask`, provides the intermediate values of the ranges between the setting `d.current` and the setting `d.target`. Using a `TaskProxy` (rather than a plain task) has the benefit that restarting it while playing stops the previous cross-fade task and begins again seamlessly from the current interpolated setting. Finally, the method `d.fadeTo` allows for fading from any stored setting to any other, with arguments for `fadetime` and `autostop`. This completes the functions necessary for creating a cloud that continually evolves based on given specifications.

```
(
   // and some parameters for controlling the fade
d.stopAfterFade = false;
d.xfadeTime = 5;
d.morphtask = TaskProxy({
   var startSet = d[\current], endSet = d[\target];
   var stepsPerSec = 20;
   var numSteps = d.xfadeTime * stepsPerSec;
   var blendVal, morphSettings;

   if (d.target.notNil) {
        (numSteps).do {|i|
        //   ["numSteps", i].postln;
              blendVal = (i + 1) / numSteps;
              morphSettings = endSet.collect({|val, key|
                   (startSet[key]? val).blend(val, blendVal)
              });
              d.current_(morphSettings);
              (1/stepsPerSec).wait;
        };
        d.current_(d.target.copy);
        "morph done.".postln;
        if (d.stopAfterFade) {Tdef(\cloud0).stop;};
   };
}).quant_(0);    // no quantization so the task starts immediately

/* test morphing
(
Tdef(\cloud0).play;
d.target = d.set6.copy;
```

```
d.morphtask.play;
)
Tdef(\cloud0).stop;

   // playing a finite cloud with tendency mask:
(
Tdef(\cloud0).play;         // begin playing
d.stopAfterFade = true; // end cloud when crossfade ends
d.xfadeTime = 10;   // set fade time
d.target = d.set8.copy;  // and target
d.morphtask.play;    // and start crossfade.
)
*/
// put fading into its own method, with optional stop.
d.fadeTo = {|d, start, end, time, autoStop|
   d.current = d[start]? d.current;
   d.target = d[end];
   d.xfadeTime = time? d.xfadeTime;
   if (autoStop.notNil) {d.stopAfterFade = autoStop};
   d.morphtask.stop.play;
};
/*    // tests fadeTo:
Tdef(\cloud0).play;
d.fadeTo(\current, \set2, 20);
d.fadeTo(\current, \set6, 10);
d.fadeTo(\current, \set5, 3, true);
Tdef(\cloud0).play;
d.fadeTo(\current, \set1, 3, false);
*/
);
```

Figure 8.26
Cross-fading between different settings with a `TaskProxy`.

The final addition to this example is a graphical user interface (GUI). Developing this was an incremental process, as before; however, rather than maintaining the intermediate steps of a GUI, it is much easier to pack the GUI creation into a single function, add more elements, and run the function again when more details have been added. For just-in-time programming, we have come up with the notion of *throwaway GUIs,* which should be very easy to make and should never influence the state of what they display (like all GUIs), so they can be opened and closed at any time; they should require little writing effort, have a small footprint in the rest of the code, and should spend little and constant computational effort on displaying what they display. Thus, for

such GUIs, slow update rates are fine, so that, for instance, replaying fast recorded control changes will not cause CPU spikes when they are being displayed. Seeing changes at, say, five times per second is often fully sufficient, so a tightly coupled Model-View-Controller scheme is not needed here.

In figure 8.28, passing the `Tdef` and a screen position as arguments would allow creating GUIs for multiple CloudGenMinis (though the code above would need modifications for that). To display the parameter ranges, the `EZRanger` class is used.

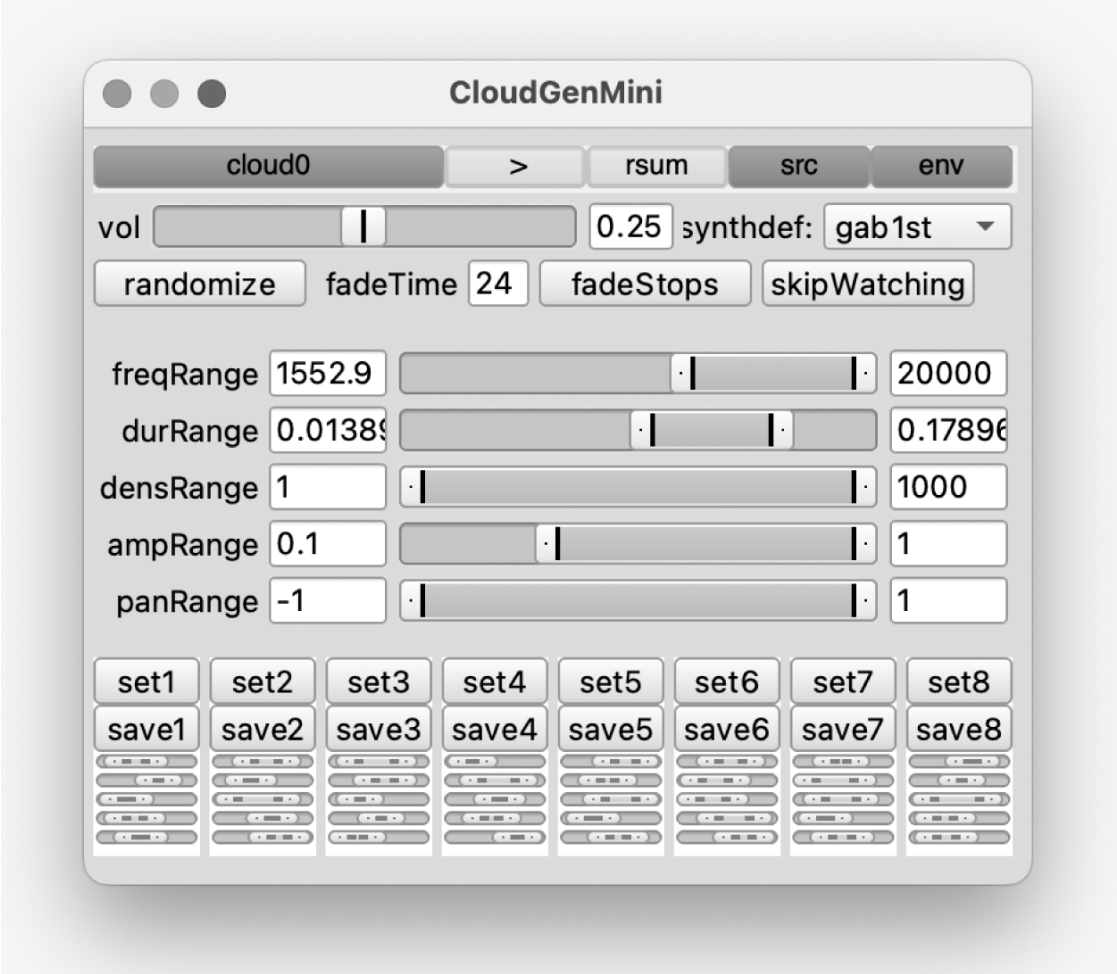


Figure 8.27
A screenshot of the `CloudGenMini` GUI //

```
(
q.makeCloudGui = {|q, tdef, posPoint|
    var w, ezRangers, fdBox;
    var setMinis, skipjack;

    posPoint = posPoint? 400@400;    // where to put the gui window
```

```
   w = Window.new("CloudGenMini",
        Rect.fromPoints(posPoint, (posPoint + (400@320)))
   ).front;
   w.view.decorator_(FlowLayout(w.bounds.copy.moveTo(0, 0)));
   w.view.decorator.nextLine;
        // a JIT-Gui for the Tdef
   TdefGui(tdef, 0, parent: w, bounds: 390@20);

/* Some extras:
   a volume slider for simple mixing,
   a popup menu for switching syndefnames;
   a button to fade to random settings, and a fadeTime box;
   a button to stop/start the skipjack for refreshing,
   so one can use numberboxes to enter values.
*/
   EZSlider(w, 245@20, "vol", \amp, {|sl|tdef.set(\vol, sl.value)},
        0.25, false, 20, 36);

   StaticText.new(w, 55@20).string_("synthdef:").align_(\right);
   PopUpMenu.new(w, Rect(0,0,80,20))
        .items_([\gab1st, \gabWide, \percSin, \percSinRev,   \percN
oise])
        .action_({|pop| tdef.envir.synName = pop.items[pop.   valu
e]});
   w.view.decorator.nextLine;

   Button.new(w, 90@20).states_([[\randomize]])
   .action_({
        tdef.envir.target_(d.randSet);
        tdef.envir.morphtask.stop.play;
   });

   fdBox = EZNumber.new(w, 90@20, \fadeTime, [0, 100, \amp, 1],
         {|nbx| tdef.envir.xfadeTime = nbx.value},
        tdef.envir.xfadeTime, false, 60, 30);

   Button.new(w, 90@20).states_([[\continuous], [\fadeStops]])
        .value_(tdef.envir.stopAfterFade.binaryValue)
        .action_({|btn|
             tdef.set(\stopAfterFade, btn.value == 1)
        });

   Button.new(w, 90@20).states_([[\skipWatching], [\skipWaiting]])
        .action_({|btn|
```

```
                [{skipjack.play}, {skipjack.stop}] [btn.value].   va
lue
        });

   w.view.decorator.nextLine.shift(0, 10);

   // the range sliders display the current values
   ezRangers = ();

   q.paramRNames.do {|name, i|
        var step = [0.1, 0.00001, 0.0001, 0.0001, 0.01][i];
        var maxDecimals = [1, 5, 4, 4, 2][i];
        var ranger = EZRanger(w, 400@20, name, q.paramNames[i],
               {|sl| tdef.envir.current[name] = sl.value;},
        tdef.envir.current[name], labelWidth: 70, numberWidth: 50,
              unitWidth: 10);
        ranger.round_(step);
        ranger.hiBox.minDecimals_(0).maxDecimals_(maxDecimals);
        ranger.loBox.minDecimals_(0).maxDecimals_(maxDecimals);

        ezRangers.put(name, ranger);

   };

              // skipjack is a task that survives cmd-period:
              // used here for lazy-updating the control views.
   skipjack = SkipJack({
        q.paramRNames.do {|name| ezRangers[name].value_(tdef.   env
ir.current[name])};
        fdBox.value_(tdef.envir.xfadeTime);

        // mark last settings that were used by color?
        // a separate color when changed?

   }, 0.5, {w.isClosed}, name: tdef.key);
   w.view.decorator.nextLine.shift(0, 10);

   // make a new layoutView for the 8 presets;
   // put button to switch to that preset,
   // a button to save current settings to that place,
   // and a miniview of the settings as a visual reminder in it.

        // make 8 setButtons
   tdef.envir.setNames.do {|setname, i|
```

```
        var minisliders, setMinis;
        var zone = CompositeView.new(w, Rect(0,0,45, 84));
        zone.decorator = FlowLayout(zone.bounds, 0@0, 5@0);
        zone.background_(Color.white);

    Button.new(zone, Rect(0,0,45,20)).states_([[setname]])
        .action_({
            // just switch:
    // tdef.envir.current.putAll (d[setname]? ())
            tdef.envir.target = tdef.envir[setname];
                tdef.envir.morphtask.stop.play;
    });

    Button.new(zone, Rect(0,0,45,20))
        .states_([["save" ++ (i + 1)]])
        .action_({
            d[setname] = tdef.envir.current.copy;
            setMinis.value;
        });
    minisliders = q.paramRNames.collect {|paramRname|
        RangeSlider.new(zone, 45@8).enabled_(false);
    };
    setMinis = {
        q.paramRNames.do {|paramRname, i|
            var paramName = q.paramNames[i];
            var myrange = d[setname][paramRname];
            var unmapped = paramName.asSpec.unmap(myrange);
            minisliders[i].lo_(unmapped[0]).hi_(unmapped[1]);
        }
    };
    setMinis.value;
    };
};
q.makeCloudGui(Tdef(\cloud0))
);
```

Figure 8.28
A lightweight GUI for `CloudGenMini`.

`Tdefs` have a GUI class, `TdefGui`, that can be placed in a window like a single view. A `TdefGui` shows a number of aspects of its `Tdef`: name, playing state, and whether the `Tdef` has a source and an environment. Clicking on the `src` and `env` buttons opens text windows for editing their code.

A “Randomize” button cross-fades from the current setting to a new random setting in the cross-fade time displayed in the number box next to it.

Slow display updating is handled by a `SkipJack`: every 0.5 second, while the window exists, it displays the parameter ranges in `d.current` in the appropriate views, and it updates the `xfadeTime` box if the fade time has changed. This could be optimized by caching previous states (e.g., as done in `TaskProxyGui`), but it was not deemed necessary here.

The next section in `q.makeCloudGui` creates three elements for each stored setting: a button to cross-fade to that setting; a button to store the current setting at that location; and miniature visual reminders of the setting stored in each location.

Though it is not ideal that these mini-displays are updated only when settings are stored with the buttons, it was considered acceptable for a sketch; experimenting with other features was more interesting.

The bottom line of GUI elements provides volume control, switching between SynthDefs, and toggling `stopAfterFade` mode. Finally, there is a button for turning `the SkipJack` off, so one can write into the number boxes. (Currently, the `SkipJack` always writes the current values back into the GUI without checking whether one is typing. Though checking is possible, coding it is more effort than turning the `SkipJack` on and off when needed.)

8.8 Conclusion

Objects and object models are wonderful means to experiment with concepts, and SuperCollider supports elegant ways of developing ideas fluidly. As the Smalltalk tradition maintains, exploring ideas interactively by writing code is a great way to think through code and gradually understand more deeply what one wants to achieve. By experimenting with different working approaches and coding styles such as those described here, one can study the art of maintaining a flow of evolving ideas, making things efficient where necessary, and keeping things flexible wherever possible.

Notes

1. See https://iterati.net/pbup/.
2. See http://sonenvir.at/data/lattice.

References

Alpert, S. R., K. Brown, and B. Woolf. 1998. *The Design Patterns Smalltalk Companion*. Reading, MA: Addison-Wesley.

Beck, K. 1996. *Smalltalk Best Practice Patterns.* Upper Saddle River, NJ: Prentice Hall.

Chaitin, G. 1998. *The Limits of Mathematics*. Singapore: Springer.

Dannenberg, R., D. Rubine, and T. Neuendorffer. 1991. “The Resource-Instance Model of Music Representation.” In *Proceedings of ICMC 1991*, pp. 428–432. Montreal: International Computer Music Association.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Hofstadter, D. 1995. *Fluid Concepts and Creative Analogies.* New York: Basic Books.

Iverson, K. E. (1979) 1980. “Notation as a Tool of Thought.” ACM Turing Award Lecture. *Communications of the ACM*, 23(8): 444–465.

Kay, A. C. 1993. “The Early History of Smalltalk.” In *Proceedings of the 2nd ACM SIGPLAN Conference on the History of Programming Languages*, pp. 69–95. New York: ACM.

McCartney, J. 2002. “Rethinking the Computer Music Language: Super Collider.” *Computer Music Journal,* 26(4): 61–68.

Smith, B. C. 1996. *On the Origin of Objects*. Cambridge, MA: MIT Press.

III EDITORS AND GUI

9 Installing, Setting Up, and Running the SuperCollider IDE

Norah Lorway

Introduced in version 3.6, the SuperCollider IDE (Integrated Development Environment) is a cross-platform code development environment which has been developed for use with SuperCollider. This chapter gives an overview of installing, setting up and running the SuperCollider IDE. The origins of the IDE are discussed along with in-depth descriptions of its components and uses.

9.1 Background and History of SuperCollider Development Environments

SuperCollider is influenced by Smalltalk (Leban et al 2013). One of the features that it shares with Smalltalk is tight integration between the interpreter and the development environment (Leban et al 2013). The first development environment for SuperCollider was known as *SC.app* (McCartney 2002), which was developed specifically for the Mac OS and was not portable to other operating system platforms.

When the desire arose to port SuperCollider to other platforms, notably Linux and Windows, a number of approaches were developed, often based on plug-ins or extensions to existing text editors. To port SuperCollider to Linux, Stefan Kersten created `scel`, a SuperCollider editor mode for Emacs (Kersten and Baalman, 2011). This editor mode was one of the most feature-rich solutions for Linux, as it supported syntax highlighting, some introspection, support for the older HTML-based help system, as well as some limited form of method call assistance (Leben et al 2013).

Other code editor extensions have since been developed as part of the official SuperCollider distribution. These include *scvim* (for vim), *sced* (for gedit) and *scate* (for Kate) (Leben et al 2013). Some of these also served as alternative editors on MacOS. In addition to these, several bespoke code environments have been developed for the use of SuperCollider with Windows, etc., some of them cross-platform. These include *qcollider* (a Qt-based editor), *scfront* (a tcl/Tk-based editor) and *PsyCollider* (Fraunberger 2011), a Python-based editor that had been originally distributed with the Windows port of SuperCollider (Leben et al 2013). Not all of these are still in active development. For a more detailed discussion of alternative SC editors, see chapter 10.

9.1.1 Motivations for the New IDE

A few factors motivated the creation of a new, cross-platform SuperCollider IDE. Many of the text editors mentioned in the previous section are based on editors which are geared toward power users and experienced programmers, which are often not very accessible for beginners, or users with a background more focused in music. This could have a gatekeeping effect and limit inclusivity.

The lack of a unified cross-platform coding environment was also a disadvantage because it limited knowledge sharing between users of different environments. In terms of upkeep, each programming environment had to be maintained separately. This involved considerable duplication of effort and required significant work by developers to provide even a roughly similar experience across different environments.

In 2011, Blechmann et al began work on a new IDE dedicated solely to SuperCollider. The goal of this IDE was to address the above issues and to bring about a unified user experience across all supported platforms, making the IDE a more accessible gateway into SuperCollider while retaining some "power features" valued by experienced users familiar with mature editors like Vim or Emacs (Blechmann 2011) and their SC-specific extensions.

9.2 Overview of the SuperCollider IDE

9.2.1 Graphical Interface

The IDE is implemented using the Qt GUI framework,[1] which provides a consistent GUI appearance and behavior across all supported platforms. Figure 9.1 shows an annotated view of the default appearance on MacOS, though Windows and Linux users will find the appearance to be very similar.

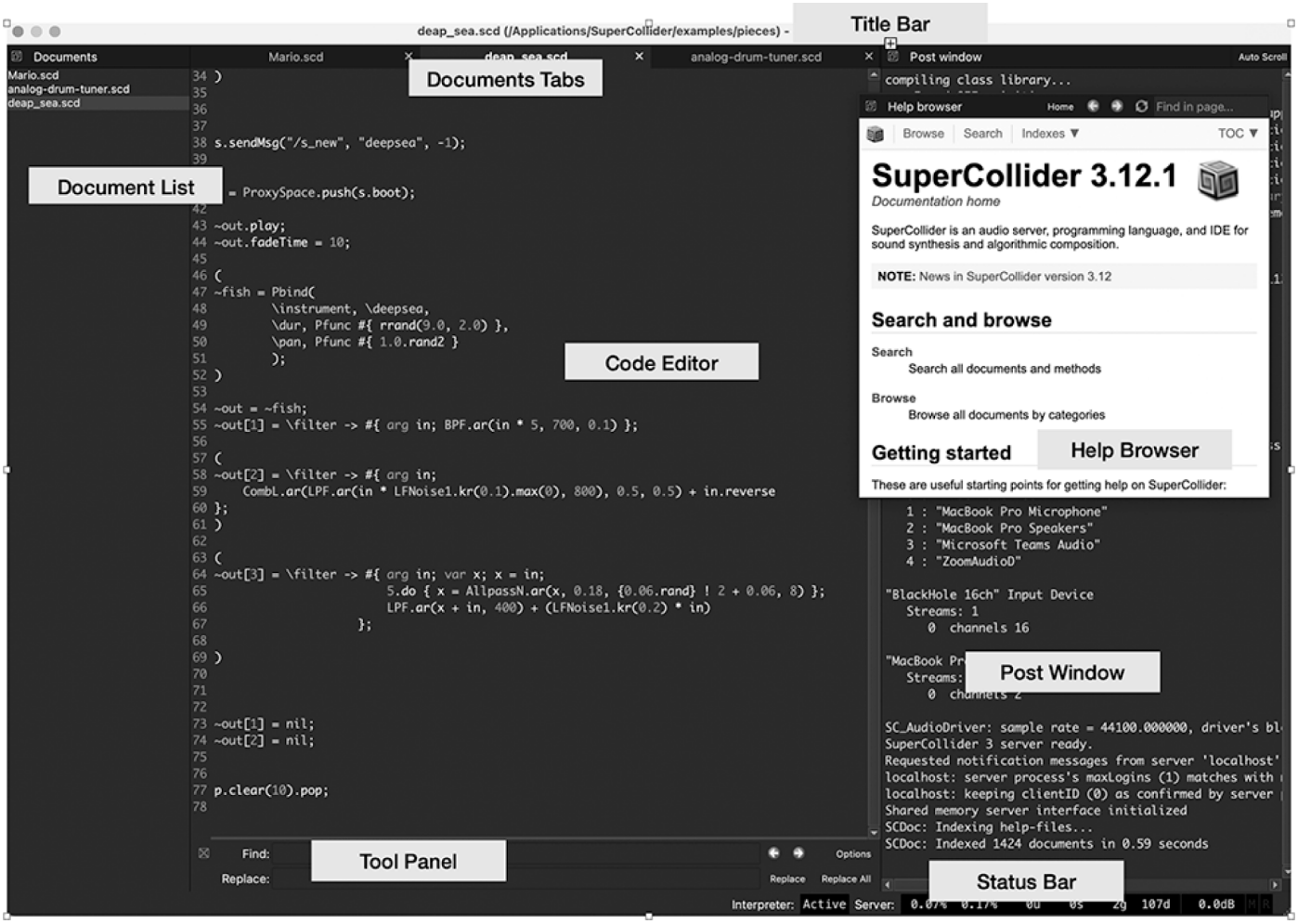


Figure 9.1
SuperCollider IDE on MacOS.

9.2.2 Central Components Overview

The following section describes the main components of the IDE. The IDE displays a single main window featuring a *code editor* widget as the central feature. The *title bar* displays the current session name, along with the file name of the current document in use. (A session is a user-defined set of files and workspace layout. See section 9.6 for a discussion of IDE sessions.) The IDE uses *document tabs* to allow for switching between multiple Document windows. Drop-down menus can be used as an alternative to tabs by setting an option in the IDE preferences. Tabs can be rearranged by dragging and dropping. The code editor can also be split horizontally or vertically to allow more than one document to be seen and edited at the same time. The code editor is discussed in more detail in section 9.4.

9.2.3 Status Bar

• The status bar, located on the bottom of the main window, is used to show the language interpreter's state, as well as that of the default synthesis server. The server

status provides a more robust and practical solution to the SuperCollider server window used in older editing environments. It shows information such as the number of running synths, CPU usage, and SynthDefs, as well as occasionally notifications related to the IDE.

• The status bar also shows elements such as *AvgCPU, PeakCPU, numUGens, numSynths, numGroups, numDefs,* and *Volume*. Figure 9.2 demonstrates where the above elements are located on the status bar:

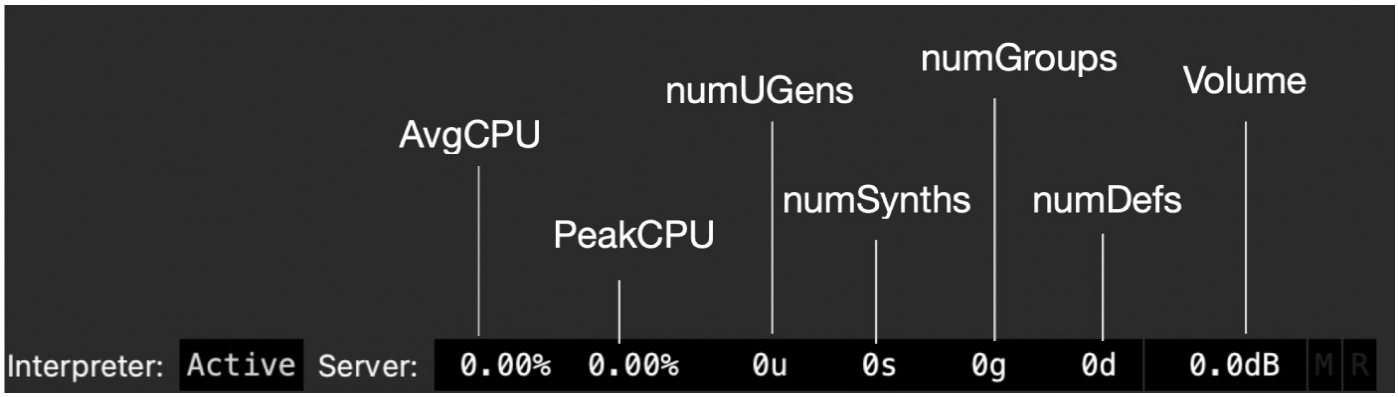


Figure 9.2
View of status bar in the IDE.

9.2.4 Docklets

Several useful features are provided as *docklets*, dockable widgets which can be snapped to the four edges of the main window around the code editor. Docklets can also be undocked and moved independent of the main window, for example to move them to a second screen. They can be shown or hidden via the menu (`View > Docklets`).

Docklets include the *help browser*, which allows for displaying and navigating through the help files; the *post window*, which displays the results of code evaluation, any responses from the interpreter, as well as various notifications, errors, and warnings that might appear; and the *documents list*, which lists all open documents for easier navigation when multiple document tabs are opened at once.

Docklets can exist in one of four different states (SuperCollider Documentation, n.d):

• `Docked`: Docklets can be snapped to the four edges of the window and can be arranged next to each other horizontally or vertically.

• `Undocked`: When a docklet is dropped outside a docking area, it will remain separate and at the top of the main windows.

• `Detached`: Docklets can be detached by clicking the top-left corner of the menu. Once the docklet is detached, it will behave in the same way as a normal window and can be minimized and maximized.

• Hidden: A docklet can be hidden using the menu in the top-left corner of the docklet.

9.2.5 Help Browser

SuperCollider's help browser is provided as a docklet. All the help options are accessible under Help in the main menu. The default keyboard command Ctrl+D is the most direct way to access the help browser while the user is coding. This will navigate the Help Browser to a page connected to the text near the cursor in the following situations:

• If the cursor is on a class or method name, the associated documentation will be displayed.
• Otherwise, the help browser shows the results of a text-based search over the documentation files.

The Ctrl+D shortcut will work in several places in the IDE. These include the Editor, Command Line, Post Window and the Help Browser. Using the Ctrl+Shift+D shortcut will allow the user to go directly to a specific help page, showing a pop-up search box which allows the user to type the name of a method or a class.

9.2.6 Tool Panels

Tool panels display various tools related to code editing. The panels are hidden by default and are displayed only when one of the tools are triggered. These can be opened via View > Tool Panels menu action, as well as by using assigned keyboard shortcuts.

Tools include the following:

• Command line: The command line allows the user to evaluate one-line SuperCollider expressions quickly. The history of evaluated expressions can be navigated using the Up and Down arrow keys, where the Up arrow shows older expressions and the Down one shows more recent ones.
• Find/Replace: This is a standard tool for finding and replacing text in a document. It supports regular expressions and back references. The Find tool is used to locate text in an open document. When text is typed into the Find field, it is searched for and the results are highlighted, with the first found result selected. Users can navigate the results of the search by pressing the Return key, which will allow them to cycle through all the results.
• Replace area: This is an extension to the Find panel and adds an additional field where text can be entered in order to replace the search results. This can be done by pressing the Return key in the Replace field, which will then replace the current search result allowing the next result to be selected. Replace All can be used to replace all the search results; the status bar keeps track of these and reports the number of replacements. Search options can be accessed by clicking on the button in the top-

right corner of the `Find/Replace` panel. Options include `Match Case`, which makes the search case sensitive; `Regular Expressions`, a powerful way to search for any text that potentially matches a pattern (IEEE 2017); and `Whole Words`, which specifies whether the search includes instances of the query string that occur within a word or identifier—for example, whether the query "Collide" matches the word "SuperCollider" or only the word "Collide" appearing in isolation (SuperCollider Documentation, n.d.).

• `Go-To -Line`: This allows the user to quickly jump to a specified line number in the current document.

9.3 Starting and Running the System

9.3.1 System Control

As noted in chapter 1, two secondary applications are used when working with SuperCollider: *the language interpreter* and *the audio server*. The language interpreter is concerned with interpreting and executing any SuperCollider code and is started automatically with the IDE. It can be stopped and restarted via the `Language` menu or with shortcuts. This is useful particularly in scenarios where the code gets stuck in an infinite loop or if there is an interpreter crash. The box in the status bar which is labeled "Interpreter" will display the word "Active" in green when the interpreter is running. Under normal use, there should be no reason to restart the interpreter; however this can be done from the Language menu if needed.

The audio server is what allows the SuperCollider code to produce sound. Unlike the language interpreter, the audio server does not start automatically with the IDE and must be started using code, the `Server->Boot Server` menu item, a keyboard shortcut (`Ctrl/Cmd + B`), or through the pop-up menu available by clicking the status bar, as shown in Figure 9.3.

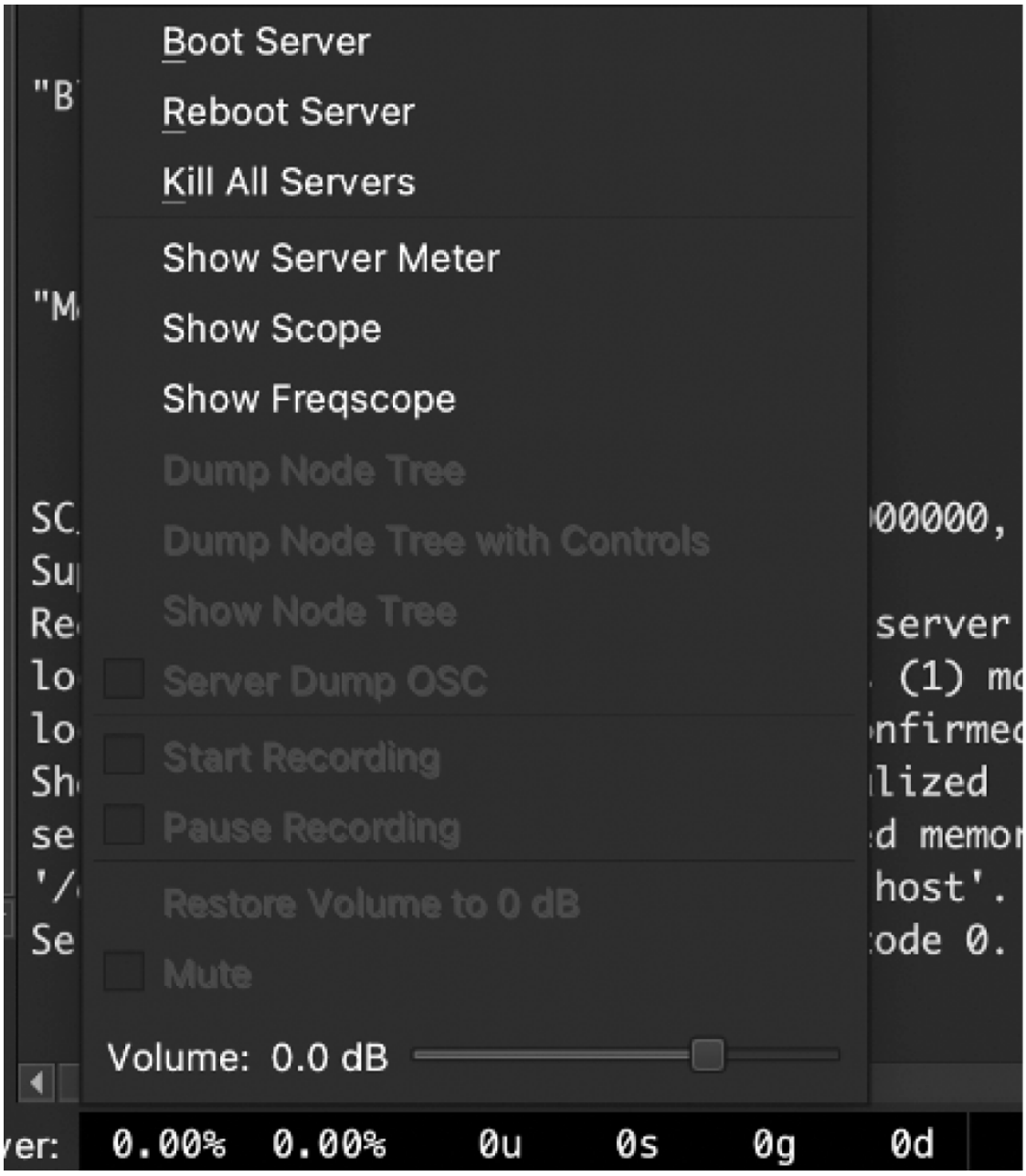


Figure 9.3
Starting up server through the status bar.

The language interpreter must be running already for the audio server to start. When the audio server is running, the status bar box labeled “Server” will display the status in green. (see 9.2.3 above). The status bar pop-up menu also contains other useful audio-related options, such as ways to dump the node tree and to show sound meter levels while audio is playing, as well as options to kill or reboot the server in case of problems. To access this functionality, simply click on the audio server status box (Leben et al. 2013).

9.4 Code Editing

The SuperCollider IDE includes helpful code-editing assistance features like those typically offered in popular IDEs used for most widely used programming languages. These include syntax highlighting, automatic code completion, automatic indentation, and method call assistance.

9.4.1 Syntax Highlighting

The SuperCollider IDE updates its syntax highlighting on the fly, taking advantage of the strict lexical rules adhered to by the SuperCollider language compiler. This is in contrast to existing SuperCollider editor extensions, which would normally reuse generic support of their host editors, often only updating highlighting on "explicit request via the user interface" (Leben et al. 2013).

9.4.2 Automatic Code Completion

Automatic code completion, or autocompletion, works in such a way that as code is typed, the editor will assist the user in providing a set of suitable continuations of code in a pop-up menu based on the context. This is a helpful aid when trying to write code quickly, particularly in live performance settings such as while live coding, giving the user a reminder of the valid identifier names in a given context as well as reducing the possibility of typing errors.

The following cases allow auto completion (<…> indicates possible additional typing to refine the selection list):

```
1. Class Names
    Sin <…> (or something similar)
```

Typing three letters (as in the example in figure 9.4) of a class name will populate a list of classes beginning with the first three letters typed.

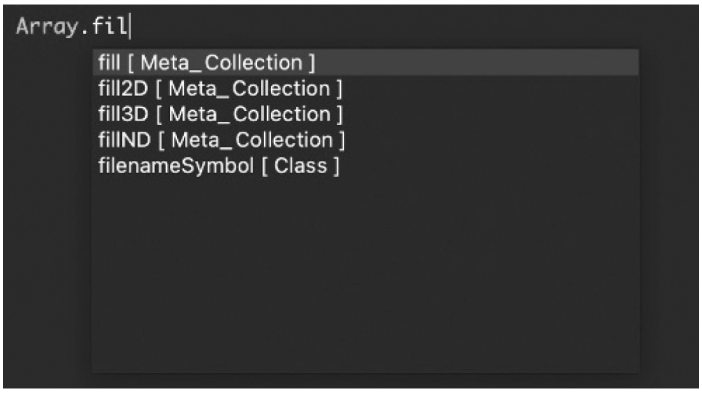


Figure 9.4
Automatic code completion in SuperCollider.

```
2. Method names following class names
```

```
Array.<…>
```

When there is a dot is typed after a class name, a list of methods of that class will populate.

3. Method names following literals and built-ins

```
123.<…>
currrentEnvironment.<…>
```

These are completed from the set of instance methods from when a dot is typed after a literally (number, string, symbol) or from a built-in (`currentEnvironment`, `thisProcess`, etc.).

4. Method names following a variable name

```
func.<…>
```

In this case, the class of the variable can be determined only when the code is executed (variables in SuperCollider are untyped); thus, the method is completed from a list of all methods of all classes, starting with the first three letters typed (see figure 9.3).

Typically, automatic completion of methods of known classes will populate immediately once the dot (".") is typed in the IDE. The only exception to this is in the case of methods using integer literals. In this scenario, automatic completion will begin only after one character is typed; otherwise, a redundant completion is triggered on a dot in a float literal (Leben et al. 2013).

When autocompletion appears, the user can use either the `arrow keys` or `Page Up` and `Page Down` to select an option. Pressing the `Return` key will insert the option into the code. It should also be noted that the completion menu will disappear if the currently typed text matches one of the autocompletion options exactly. If the auto completion menu is not shown, it can be triggered by typing `Ctrl+Space`. Finally, it can be dismissed by pressing the `Escape` key.

9.4.3 Automatic Indentation

SuperCollider's IDE automatically indents code as it is typed according to opening and closing parentheses. As the user types the closing brackets, the indentation will decrease to line up with the opening brackets.

If there is something wrong with the indentation of the code, pressing the `Tab` key will fix this by automatically indenting it in keeping with the surrounding brackets.

`Indentation` can also manually be inserted anywhere in the code by pressing `Shift+Tab`. Finally indentation can be configured to use either `tab characters` or `spaces`, which the user can toggle through the main menu (via `Edit` > `Use spaces`). Tabs are the default, as this was the established precendent in SC code.

The indentation of a selected region of code can also be fixed using the menu command `Edit >AutoIndent Line or Region.`

9.4.4 Method Call Assistance

Method call assistance aids users by showing a complete list of the method's argument names and their default values. This allows users to know exactly which argument they are typing and minimizes the necessity to look up documentation.

Assistance is triggered when either an opening parenthesis or a comma is typed after a method name. As in figure 9.5, a box containing the arguments will appear. In cases where the class of a method is not known, a menu of classes that implement the method will appear with the option to select one of the classes using either the arrow or the `Page Up` and `Page Down` keys.

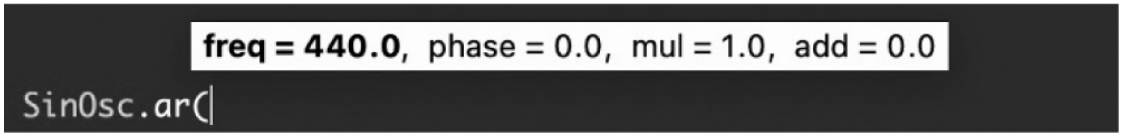


Figure 9.5
Method call assistance in the SuperCollider IDE.

For instance, in the following examples, the method call assistance will pop up immediately:

```
SinOsc.ar(
```

However, in the following instances, the class will not appear automatically and will need to be selected beforehand:

```
x.play(
```

In some special instances, the method name is not descriptive—namely, in cases where an opening parenthesis follows a class name:

```
Synth(
```

In these instances, the class method "new" is implied, and the IDE takes this into account, offering the appropriate assistance.

When assistance is triggered for a certain method call, it will remain active in the background while assistance for the method call takes place.

If method call assistance is not immediately shown, the user can trigger it by typing Ctrl+Shift+Space when the cursor is between the parentheses which surround the arguments. As with autocompletion, method call assistance can be dismissed by pressing the Escape key.

9.4.5 Editing Shortcuts

As with many development environments, the SuperCollider IDE allows code editing and help navigation to be assigned to keyboard shortcuts. In addition to the usual operations of inserting, deleting, copying and pasting text, more advanced editing actions include the ability to move or copy the current line up or down, and commands to comment or uncomment lines or sections of code. The latter can use either single-line or multiline comment syntax. Note that in contrast to standard macOS practice, not every shortcut has a corresponding menu item. These shortcuts are user editable. A complete list of actions and their currently assigned shortcuts can be viewed in the Shortcuts section of the IDE preferences.

9.5 Class Library Navigation

The SuperCollider IDE has built-in methods to help navigate the class library efficiently. Situations where this might be used include when the user wants to go directly to a particular place in a file where a certain class or method is implemented. Other scenarios include finding all locations where a class or method name has been used.

To look up implementations, press Ctrl/Cmd+I with the cursor placed on the method or class name. This will open up a dialog box with a list of locations where the method or class has been implemented. Entries can be selected by typing Return, allowing the IDE to open the file. The Ctrl/Cmd+I shortcut is not restricted to the code editor window, and also works in the Command Line, the Post Window and the Browser.

To look up references, the user should use the Ctrl/Cmd+U shortcut, which will open a dialog box like the one used when looking up implementations with the exception that it will list locations where a method or class with a certain name is used. It is useful to note that this will search only for class files within the class library.

9.6 Sessions

The SuperCollider IDE allows users to preserve the previous state of the application so that once they return to the session, it opens with the same arrangement of open documents and docklets. In order to save the current state of a session, select the Session > Save Session As menu item. Opening a session can be done by using the

`Session>Open Session` menu item. Finally, users can start a new [unsaved] session using `Session> New Session`. It is worth noting that by default, the IDE starts the last session used before closing the program. This can be changed in the IDE preferences dialog box.

9.7 Customizations

Several aspects of the SuperCollider IDE can be customized. These include colors of text, fonts, colors of the editor component, syntax highlighting, and keyboard shortcuts. The behavior of automatic indentation, as well as code evaluation, can also be configured.

The Configuration Window also allows different class library extensions to be included and excluded from compilation, and for the user to define and select presets for this. This can be helpful when working on different projects, which require different sets or versions of extensions.

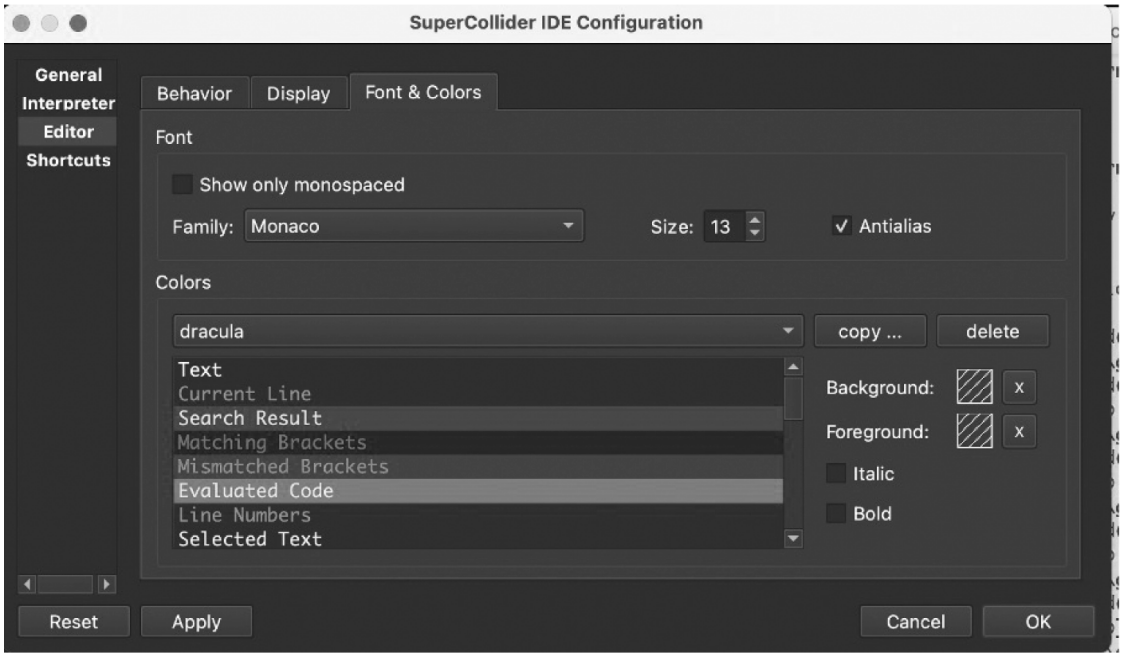


Figure 9.6
SuperCollider IDE Configuration window.

9.8 Conclusions

The SuperCollider IDE provides a useful cross-platform coding environment which integrates some popular features from other IDEs, as well as enabling useful functionalities specific to SC's language and server apps. It provides the a unified

development experience across `macOS`, Windows, and Linux, and a single code base to maintain.

The user interface is highly developed and provides a high degree of advanced coding assistance. These improvements have transformed SuperCollider into an even more powerful tool for developers and users and has also made the language even more accessible to beginner users, allowing fewer barriers to entry while facilitating knowledge exchange among users.

Note

1. https://www.qt.io.

References

Blechmann, Tim. 2011. "Supernova—A Scalable Parallel Audio Synthesis Server for SuperCollider." In *Proceedings of the International Computer Music Conference*.

Institute of Electrical and Electronics Engineers (IEEE). 2017. "The Open Group Base Specifications." Accessed April 8, 2022, from https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

Leben, J., and T. Blechmann. 2013. "SuperCollider IDE: A Dedicated Integrated Development Environment for SuperCollider." In *Proceedings of Linux Audio Conference*.

McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal*, 26(4): 61–68.

SuperCollider Documentation. "SuperCollider IDE." Accessed November 17, 2021, from https://doc.sccode.org/Guides/SCIde.html.

10 Alternative IDEs for SuperCollider

Konstantinos Vasilakos

10.1 Introduction

This chapter presents an overview of available Integrated Development Environments (IDEs) for running and editing SuperCollider code. While this chapter is not a technical appraisal with comparative analysis of each IDE, it aims to provide a practical view of available editing platforms, and more specifically to provide information about available extensions which might come in handy while running SuperCollider in an alternative IDE. Examples of editors are outlined and a closer look is given to GNU Emacs as a case study of a powerful alternative IDE for SC.

Though SuperCollider's own native IDE was introduced in version 3.6, alternative code editors allow a higher level of customization when communicating with the language's interpreter. While some customization is also possible in the native IDE of SuperCollider (such as font selection, modification of keyboard shortcuts, and application of alternative color themes), some more specialized IDEs allow for a higher level of personal adjustments for coding. These include injecting code snippets in workspace to advance the implementation speed of common tasks, through to more specialized uses, such as supporting SuperCollider code blocks within other document formats, with authoring manuals and technical documents only a few keystrokes away. There is also the ability to divide workspaces for different programming languages, avoiding the need to use different editors to communicate with multiple pieces of software in real time. One way that this might prove useful is better visual representation of incoming data interacting with SuperCollider. In addition, handling or housekeeping large projects with more efficiency, as well as integrating with version control from inside the code editor, are just some of the numerous conveniences that are offered while working with an alternative IDE.

Thus, alternative IDEs provide an environment to establish your idiomatic preferences while coding, interacting with other packages that offer support for programming and software development.

10.2 Overview of IDEs for SuperCollider

A wide range of IDEs supporting an array of programming languages are available, such as Jet Brains, Sublime Text, (Neo)Vim, Visual Studio Code, and Emacs. Of these, Visual Studio Code, (Neo)Vim, and Emacs are the most commonly used for SuperCollider. All allow interaction with the language's interpreter, and from there to the SuperCollider server synthesis engine. As a well-supported IDE, each also enables real-time integration with third-party packages or plug-ins (depending on a given platform's terminology). Links for each platform and its SuperCollider-specific extension sets can be found in table 10.1.

Table 10.1
Popular alternative IDEs for SuperCollider, with cross-platform links for installation of the basic IDE and SuperCollider specific extension

IDE	URL for IDE (Windows, Linux, Mac)	SuperCollider specific extension package	URL for SuperCollider specific extension set
Neovim	https://github.com/neovim/neovim/wiki/Installing-Neovim#install-from-download	Scnvim	https://github.com/davidgranstrom/scnvim
Vim	https://www.vim.org/download.php	Scvim	https://github.com/supercollider/scvim
VS Code	https://code.visualstudio.com/download	scvsc, vscode-supercollider, vscode-supercollider (LSP)	https://marketplace.visualstudio.com/items?itemName=scvsc.scvsc https://marketplace.visualstudio.com/items?itemName=jatinchowdhury18.vscode-supercollider
VSCodium (open-source VS Code)	https://vscodium.com/#intro		
Emacs	https://www.gnu.org/software/emacs/download.html	Scel	https://github.com/supercollider/scel

According to a survey[1] on Stack Overflow, Visual Studio Code has been very popular among programmers in recent years. Visual Studio Code provides a robust and user-friendly interface for installing packages and their configuration, all in one package. Emacs, on the other hand, is one of the older players in the realm of text editors, developed in the C and Emacs Lisp languages. Vanilla Emacs might appear "old school," but many configuration frameworks are available to supercharge the editor (as discussed in section 10.4.8). It provides a fully customizable and self-documented environment, although it may require some Elisp programming skills to advance bespoke functionality. Although this may appear to be a drawback, especially when compared to the other editors mentioned above, it can also be an asset, allowing users to develop superpowers to unlock the full system; deep idiomatic preferences can reach

advanced levels of interaction, with Emacs almost its own mini-operating system. Emacs can provide a very powerful tool for software development at the same time that it allows the organization of day-to-day tasks and creative coding as a pastime.

Table 10.1 provides an overview of the most popular[2] alternative IDEs and their SuperCollider package names. Each editor and their packages provide up-to-date guides on installation according to each operating system, with URLs in the table correct at the time of writing.

All these IDEs are supported by numerous external packages that may expand the workflow while coding, such as those for text highlighting and syntax error handling in real time. Most of the time, these extensions may be installed using built-in package managers, even if some additional tweaking is required to SuperCollider installation settings.

In order to start using SuperCollider with another IDE, some platform-specific steps are required. Depending on the IDE, this may be as straightforward as declaring the path to the executable files of the SuperCollider language and server, but sometimes it may require additional work. The most up-to-date guidance on the use of the packages is available with the packages themselves, from installation information to crucial functionality such as key bindings for each action (booting the server, compiling the class library, and other relevant tasks). Of course, additional packages for a given IDE are potentially available, with SuperCollider then benefiting from, for instance, syntax highlighting, error handling, and the integration of other libraries. Figure 10.1 illustrates scnvim in Nvim running in a terminal window.

In the following sections, we will look at two example alternative editors: first Visual Studio Code briefly, and then Emacs in more depth.

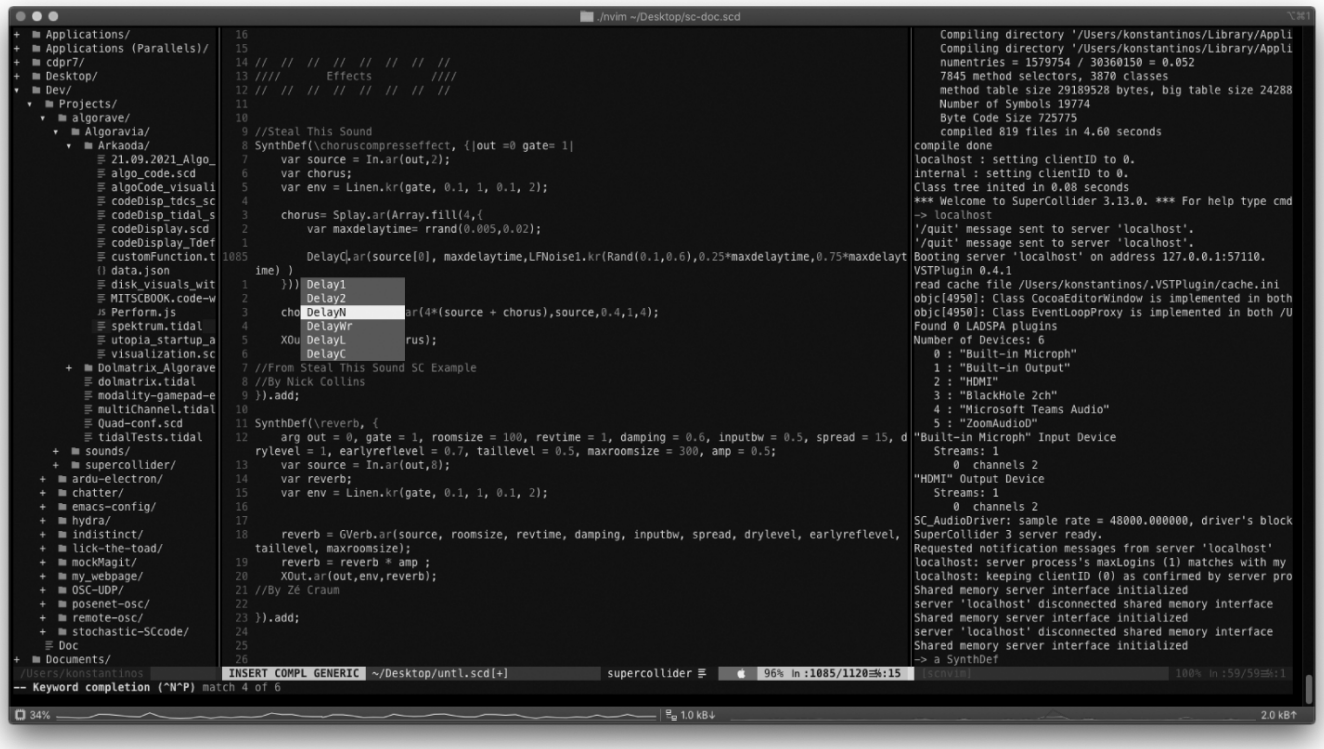


Figure 10.1
scnvim extension for SuperCollider in Nvim.

10.3 Visual Studio Code

These instructions refer to the most recent version of VS Code at the time of writing, but readers may need to navigate changes in the software over time. There are three available extensions at the time of writing of this chapter. These are *scvsc, vscode-supercollider,* and another *vscode-supercollider*, which implements a language server/client and communicates using the Language Support Protocol (LSP). The basic *vscode-supercollider* extension does not offer language interaction, but instead always wraps code in an `s.waitForBoot` function to make a command for the sclang executable run through the command line. On the other hand, *vscode-supercollider* with Language Server protocol support resembles the native IDE of SuperCollider, providing language services and tasks, such as autocompletion, syntax highlighting, code analysis, etc. LSP was developed by Microsoft, but it is offered as an open protocol for communication between a language server and an IDE. While the server is a separate program which understands the language's specifics, it communicates with the client IDE through the LSP to facilitate language-related services. A thorough explanation of the LSP is beyond the scope of this chapter; For further details about the architecture and the implementation of LSP in VS Code, see this link.[3] A recommended

current working extension is *scvsc*, which offers region evaluation and execution of highlighted code in real time and post window monitoring in the output tab of the editor.

To install an extension in VS Code, click the activity bar leftmost in the editor, click on the Extensions icon, and in the search field look for *scvsc*. If there is no activity bar appearing on the window, press Cmd+Shift+X (Mac)/Ctrl+Shift+X (PC) to bring up the Extensions tab in the editor. Click Install. After installation is completed, Disable and Uninstall buttons will appear. In the main view of the window, you will find instructions for both MacOS and Windows operating systems. Scroll down for instructions and features of the extension. The configuration file for *scvsc* is stored in ~/Library/Application Support/Code/User/settings.json in MacOS, %APPDATA%\Code\User\Settings.json for Windows, and ~/.config/Code/User/settings.json for Linux. This file contains the path to the sclang executable. For MacOS, add the path /Applications/SuperCollider/sclang; for Windows, C:\\Program Files\\SuperCollider-3.9.3\\sclang.exe\; and for Linux, /usr/local/bin/sclang. While this setting is required by all packages, the second *vscode-supercollider* (LSP) requires additional tweaking and installation of the current developer build of SuperCollider. The package is installed manually using the *.vsix* extension file and is available at the link in table 10.1. After downloading, it can be added in VS Code using the Command Palette; to bring up this window in VS Code press CMD+SHIFT+P/CTR+Shift+P and type "Install from VSIX", select the downloaded file and press Install. After the installation, the paths of the sclang executable and an additional path located in SuperCollider's user's directory are required. In MacOS, this is ~/Library/Application Support/SuperCollider/sclang_conf.yaml; in Windows C:\Users\<USERNAME>\AppData\Local\SuperCollider; and ~/.config/SuperCollider/sclang_conf.yaml in Linux. It also requires the Language Server Quark, which comes with the package, to be installed in SuperCollider. The installation occurs through the Command Palette in VS Code (Cmd+Shift+P/Ctr+Shift+P), and typing "SuperCollider: Update Language Server.quark."

Settings such as paths can be configured using two ways: from Open User Settings on the Command Palette or by editing the JavaScript Object Notation file. If SuperCollider is installed in a different location than suggested, you must configure the path accordingly. Refer to the instructions for more details for each operating system. Inside a SuperCollider file (.scd) as open in VS Code, bring up the Command Palette to initialize SuperCollider and other available commands. A screenshot of running SuperCollider using *scvsc* in VS Code is provided in figure 10.2.

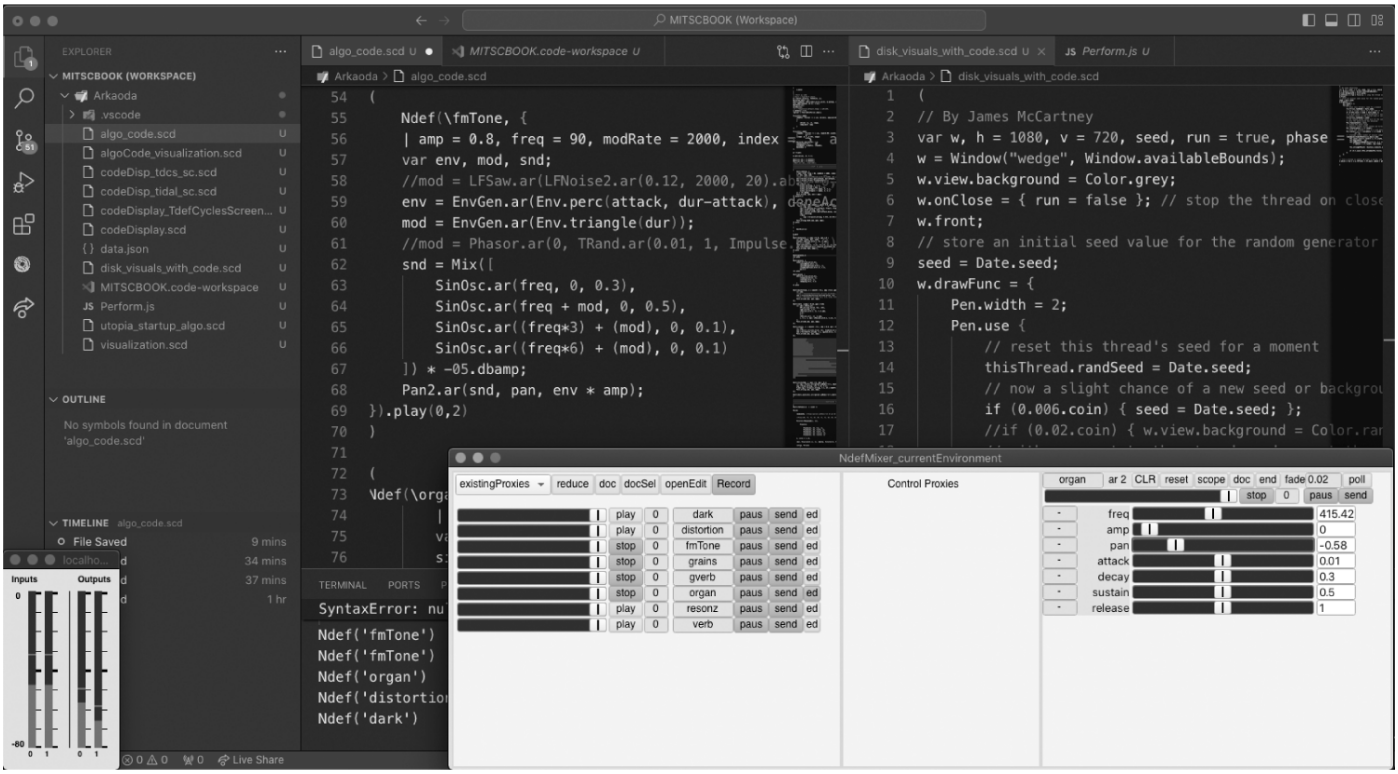


Figure 10.2
scvsc and SuperCollider in VS Code.

In VSCodium, some packages are missing when searching in the Extensions manager, and thus some extensions may not be installed in the same straightforward way as in vanilla VS Code. A workaround to install missing extensions is to download the file ending with the .vsix extension, which can be found at the VS Code marketplace when you click on the Download Extension tab. The installation can be done using the Install from VSIX option as previously described from the Command Palette. Alternatively, it can be installed by using the editor's command-line interface inside a terminal window in Unix/Linux environments or the command prompt on Windows. For example, the following command runs the installation of the *scvsc* extension: `$ codium–install-extension </path-to-file/scvsc.scvsc-0.1.0.vsix>`. If the path is correct, the following message will appear: "Installing extensions … (node:62911) [DEP0005] Extension 'scvsc.scvsc-0.1.0.vsix' was successfully installed."

Using a command-line interface is not unique to VSCodium. VS Code also offers many commands, though in MacOS this must be installed through the Command Palette via Install Code Command in Path. Both editors' command-line interfaces can be used for a variety of tasks from extension management to troubleshooting and status of the installation.

10.4 GNU Emacs

Emacs is one of the most powerful editors; it combines a fully customizable, extensible, and self-documented ecosystem (Petersen, 2015; Kleene, 2020; Wilson, 2021). It is also one of the older editors in the family of IDEs, dating to 1976.

Emacs is a cross-platform application, and thus all examples work across all platforms (unless stated otherwise). Chances are that you already have Emacs already installed on your machine (else go to https://www.gnu.org/software/emacs/download.html). On a Unix system, run the commands `which emacs` and `emacs--version` will reveal this. Assuming these commands ran properly you can try to open the Emacs GUI (instead of running Emacs only on the command line) by executing `emacs`. Sometimes an old version will create conflicts with newer versions that are installed by the user, and it might be necessary to unlink it and make new symlinks accordingly. While this is beyond the scope of this chapter, up-to-date information on troubleshooting and installation is available based on operating system. Emacs can be used for a wide range of tasks, from a code editing tool supporting numerous programming languages to generic software development including version control, terminal emulation, authoring technical documents in plaintext and markup languages, and literate programming.

Packages and extensions in Emacs can be installed from its built-in manager, initiated by the command [`M-x list-packages ENTER`], which will open a lengthy list of modules available in the Emacs ecosystem found in the official repository, the Emacs Lisp Package Archive (ELPA), and further locations. Further, *list-packages* will open a buffer with all available packages from which the user can select and install.

One of the attractions of Emacs that the author has found while working with SuperCollider and other languages is the ability to organize housekeeping for larger projects, while at the same time keeping all code and resource files accessible from within the same platform. Lastly, Emacs provides a built-in manual, which can be opened with the `[C-h RET]` keys. `[C-h]` is a useful key binding, as it will bring up a lot of options in the editor.

10.4.1 SuperCollider in Emacs

This section provides a case study of using SuperCollider in Emacs, giving more thorough elaboration using practical examples. This section does not aim to cover all the possible uses of SuperCollider in Emacs, but rather to illustrate various concepts and packages in connection with SuperCollider coding. SuperCollider mode in Emacs is implemented through the scel package. After configuring the paths to the executable files of SuperCollider and the mode's implementation, it can be initialized by typing `M-x` and the command *`sclang-start`*. This will compile the class library and open the SuperCollider Workspace scratch window with *sclang-mode* activated, establishing where all interaction with the interpreter takes place. To save work, a new file with extension *.scd* must be created since Emacs creates a scratch buffer only.

Once a package is installed for a particular language and configured accordingly, a mode is typically triggered by a file opened with a given extension. Init.el is the master configuration file for Emacs, found in the editor's directory. On Mac OS, this is in *~/.emacs.d/init.el*; in Windows, it can be found in C:/Users/<username>/AppData/init.el, and */ home/$USER/.emacs.d* in Linux.

At the time of writing, there is no available package for SuperCollider in default databases and installed in the default way, such as through ELPA. The project is available in GitHub from SuperCollider's scel (currently at version 1.0.0) repository kindly maintained to this date. The most straightforward way currently is to install the `scel` Quark through the built-in extension manager of SuperCollider. First and foremost, in order to make SuperCollider run in Emacs, some paths must be declared in user configuration files. Below is an outline of steps for configuration.

10.4.2 Notes for Setting SuperCollider in Emacs

Installation steps for cross-platform installation below are reproduced from the `scel` project in Github from which the preferred method is using SuperCollider's package manager Quarks.

10.4.2.1 Install scel on Mac OS

1. Run `Quarks.install("https://github.com/supercollider/scel");x`
   In the post window, something like this will appear:

```
> Installing scel
> Adding path: ~/Library/Application Support/SuperCollider/downloaded-quarks/scel scel installed
> Quark: scel[1.0.0]
```

2. Execute `Quarks.folder.postln;` from the post window, copy the invoked path, such as

```
> "~/Library/Application Support/SuperCollider/downloaded-quarks/scel"
```

3. In your Emacs configuration file, add the /el subdirectory at the end of your path:

```
(add-to-list 'load-path "~/Library/Application Support/SuperCollider/downloaded-quarks/scel/el")
```

And add the two lines below too:

```
(add-to-list 'exec-path "/Applications/SuperCollider.app/Contents/M
acOS")
(require 'sclang)
```

10.4.2.2 Install scel on Debian Linux This is tested on the Debian version for Raspberry Pi (3). First, update and then install it with `sudo apt-get install supercollider`. Follow steps 1, 2, 3, and 5 from section 10.4.2.1 and assign the Emacs variables of scel Quark path apropos to Linux OS, such as

```
(add-to-list 'load-path "/home'$USER/.local/share/SuperCollider   /
downloaded-quarks/scel/el)
```

10.4.2.3 Installing Emacs and scel on Wındows Open the downloaded .exe file and run through the installation steps as prompted, including where Emacs will be installed (e.g., C:\Program Files\Emacs), and press Install to start the installation process. Once installation is through Emacs will appear in your newly installed applications. Open Emacs and type `[M-x d ~/]` to open your home folder `(C:/Users/<username>/AppData/Roaming/.emacs.d:)` and click on the .emacs.d directory.

Create init.el file if you freshly installed Emacs. Some steps on the side of SuperCollider need to be taken first. Download SuperCollider (if necessary) and open up a new file and execute steps 1, 2, 3, and 5 from section 10.4.2.1, and do not forget to configure *and assign Emacs variables of scel Quark path apropos to* Windows OS accordingly and add scel/el at the end as follows:

```
(C:\Users\<username>\AppData\Local\SuperCollider\downloaded-   quar
ks/scel/el).
```

That should be on the right track to start SuperCollider in all platforms, even if sometimes things do not work as expected due to package modifications and changes in implementations. Drawing from personal experience, online resources by dedicated communities for Emacs and scel come in very handy with up to date and cross-platform installation guides and troubleshooting.

Once configuration is complete, typing `[M-x]` (in Emacs, M stands for the Alt key), allows you to invoke the available commands.

10.4.3 Sclang Mode in Emacs

When opening a file with a specific extension, for example .scd, Emacs will switch to the associated *major* mode. This will be visible to the *modeline* (a narrow stripe lower

down on the screen, sharing the space with the *minibuffer*). This changes the behavior of the buffer accordingly. Thus, *sclang-mode* will provide access to those key bindings which are relevant to SuperCollider. Besides the major mode, which dominates the functionality and behavior of the buffer and the available mappings of keystrokes and commands, other *minor* modes may also be activated as supportive tools, for example, to highlight balanced expressions and run an autocompletion engine. Typing [`M-x describe-mode`] reveals all minor modes that are currently activated in the SuperCollider workspace.

10.4.4 Interaction with SuperCollider in Emacs

Emacs uses a couple of modifier keys as prefixes to other keys, manifestly sometimes ending in long key sequences. The prefix keys used in Emacs are *C* (Control), *M* (Meta or Alt key), and *S* (Shift). Holding down the C key while pressing another key will invoke available commands that are bound to the interpreter's functionality. For example, by default, `sclang-evaluate-region-or-line` is mapped to `[C-c C-c]`. A detailed key map of *sclang-mode* is outlined in table 10.2. A demonstration of navigating in code regions can be found at https://youtu.be/00BBNBAvKGI.

10.4.5 Basic Navigation

Fast navigation through SuperCollider code is probably the main attraction of editing documents in Emacs. Some specific keystrokes (as shown in table 10.2) are useful to memorize, especially to speed up coding in live settings, where movement in code rapidly progresses the coding process. Of these keyboard shortcuts, `[C-n/C-p]` moves between lines, and for quick code evaluation inside regions in the workspace `[C-M-n]` and p, respectively, can be used to jump between each region by using *smartparens* (a minor mode in Emacs allowing you to jump forward to the beginning of balanced expressions). A very handy key binding is executing regions enclosed in parentheses by holding the `[C-M-x]` keys. A brief video showing the evaluation of code blocks can be found at https://www.youtube.com/watch?v=66UKP-yMG4E. Finally, a favorite paraphernalia from the Emacs ecosystem that has been proven very useful for the author, especially for live coding performance, is adding the multiple cursors package,[4] which allows one to navigate and edit code in various spots in a file at the same time. A brief video of this can be found at https://youtu.be/sVnwB8m9WAA.

Table 10.2
Key bindings in sclang mode

Key	Binding
C-M-h	sclang-goto-help-browser
C-M-x	sclang-eval-defun
C-c C-c	sclang-eval-region-or-line

Key	Binding
C-c C-d	insert-date-string
C-c C-e	sclang-eval-expression
C-c C-f	sclang-eval-document
C-c C-h	sclang-find-help
C-c C-k	sclang-edit-dev-source
C-c C-l	sclang-recompile
C-c RET	sclang-show-method-args
C-c C-n	sclang-complete-symbol
C-c C-o	sclang-start
C-c C-r	sclang-main-run
C-c C-s	sclang-main-stop
C-c C-w	sclang-switch-to-workspace
C-c C-y	sclang-open-help-gui
C-c:	sclang-find-definitions
C-c;	sclang-find-references
C-c <	sclang-clear-post-buffer
C-c >	sclang-show-post-buffer
C-c [	sclang-dump-interface
C-c h	sclang-find-help-in-gui
C-c {	sclang-dump-full-interface
C-c}	sclang-pop-definition-mark
C-M-q	prog-indent-sexp
C-c C-p	prefix command
C-c C-p b	sclang-server-boot
C-c C-p d	sclang-server-display-default
C-c C-p f	sclang-server-free-all
C-c C-p m	sclang-server-make-default
C-c C-p n	sclang-next-server
C-c C-p o	sclang-server-dump-osc
C-c C-p p	sclang-show-server-panel
C-c C-p q	sclang-server-quit
C-c C-p r	prefix command
C-c C-p r n	sclang-server-prepare-for-record
C-c C-p r p	sclang-server-pause-recording
C-c C-p r r	sclang-server-record
C-c C-p r s	sclang-server-stop-recording

All key bindings can be customized and bound to any available keys. Figure 10.3 shows an example of how to customize and add your own key bindings.

```
(define-key sclang-mode-map (kbd "M-RET") 'sclang-eval-defun)   (de
fine-key global-map (kbd "M-s C") 'sclang-start)
```

Figure 10.3
User-defined keystrokes for sclang mode.

Some keys may already be bound to other commands, and checking with `[C-h k]` and trying a key sequence will reveal any overlaps.

Although long keystrokes are hard to memorize and may feel cumbersome in the beginning, especially if you are migrating from other editors, they are very advantageous in the long run. Once memorized in muscle memory, the commands can be accomplished without too much deliberation.

10.4.6 Easy Customization of SuperCollider in Emacs

The easiest way to modify the default behavior of the package is through customization of the variables interface [*M-x sclang-customize*]. Through this interface, the user can fine-tune some of the available variables, such as switching to the workspace upon initializing SuperCollider; a brief video illustrating this can be found at https://youtu.be/hhf9zPOobBU.

Emacs allows further customization in order to implement a personal workflow. Using Emacs Lisp additional commands can be implemented, and following the interactive standard, these will be available to the minibuffer.

Some examples of adding more personalized tasks in combination with existing commands follow.

Figure 10.4 defines a function which inserts the current date and time in the buffer in order to store details about when a file was last modified. It can be mapped to an available keystroke.

```
(defun insert-date-string ()
"Insert date and time string in SC"
(interactive)
(insert
"//" (current-time-string) "\n"
"//" (user-login-name) "\n"));; map this to C—c C—d
```

Figure 10.4
User-defined functions for sclang mode.

Another example, shown in figure 10.5, is a SuperCollider function including a `WhiteNoise` UGen, which is inserted in the workspace while also causing the booting of the server (https://youtu.be/GWxjqVRYAhI).

```
(defun sc-server-boot-noise-test ()
"SC Server Boot and Recompile and test sound"
(interactive)
```

```
(sclang-server-reboot)
(insert "(
s.waitForBoot{
        play{WhiteNoise.ar}
}
)"))
```

Figure 10.5
Code insertion example of user defined function for sclang mode.

All these examples could be adopted in the package's code if they were generally useful for SuperCollider. However, individual alteration of the source code of the package must be avoided since it may break after updating and create buggy behavior. Thus, the best approach to enlarge current functionality for *sclang-mode*, instead of hacking the package implementation directly, is to implement extra functions in the personal configuration file, that is, *init.el*.

Another way to tweak an existing function is through Emacs' advice and override features. Figure 10.6 illustrates how to implement similar customizations using advice; *defadvice* is used over an existing command, that is over `sclang-server-boot`, mapping `sclang-server-boot-noise` after the *before* symbol, which forces it to run before the regular function. The next line checks if the command exists and then inserts the SuperCollider function. Lastly, this must be activated exclusively.

```
(defun sc-server-boot-noise-test ()
"SC: server Boot and Recompile and test sound"
(interactive)
(sclang-server-boot)
(insert "play {WhiteNoise.ar(0.01)} "))

(defadvice sclang-server-boot (before sclang-server-boot-noise ())
"SC:insert white noise when booting server."
(if (fboundp 'sclang-server-boot)
   (insert "play {WhiteNoise.ar(0.01)} \n ")
        (message "sclang-server-boot doesn't exist")))
(ad-activate 'sclang-server-boot)
```

Figure 10.6
Other approaches to user-defined functions.

One of the few things that are hard to ignore when comparing the SuperCollider workspace in Emacs and the native IDE of SuperCollider is that the former lacks tooltips while typing and the real-time server monitoring widget. For the latter, a separate GUI window may be created with *s.makeWindow*, which will provide ongoing

information in a similar way to the native IDE. Alternatively, the server's activity can be traced from *modeline* at the bottom of the post window buffer.

10.4.7 Getting SuperCollider Help in Emacs

SC help in Emacs may be opened with `[C-c C-y]` or via executing HelpBrowser.new in the workspace, which will open an HTML format window including executable code as normal.

10.4.8 Configuration Frameworks for Emacs

Besides the vanilla version of Emacs and a plethora of packages that expand the editor's capabilities, one can also use a modern configuration framework which ships with many preinstalled modules from the Emacs ecosystem. Three such frameworks are Doom,[5] Spacemacs,[6] and Prelude.[7] These frameworks expand Emacs by offering additional functionalities to the editor, such as Vim emulation.

10.4.8.1 Doom Emacs Doom ships with several packages and helps to smooth the rough edges of the vanilla version. A selection of packages and modes can be activated by users in the file named init.el, found in the Doom (.doom.d) directory where all configuration files are located, including config.el, custom.el, and packages.el. `[C-h d h]` will open the Doom Emacs documentation. One of the main distinctions between Doom and Emacs is that all personal configuration code is stored in the config.el file in the Doom directory as opposed to the init.el file of Emacs. Doom's init.el file controls which modules are enabled. Another method for path assignment is with Doom's own `doom-load-envvars-file` configuration. On Unix-based machines with the .zshrc or bash_profile, which are used to configure the environment, one can assign paths apropos to SC executable as well as shown below. The steps are outlined in figure 10.7 using config.el and .zshrc or bash_profile files alike, depending on the user's environment for environment configuration.

```
export PATH="/Applications/SuperCollider.app/Contents/MacOS:$PATH"
export PATH="~/Library/Application Support/SuperCollider/   downloa
ded-quarks/scel/el:$PATH"
cd ~/.emacs.d/bin && ./doom env
In config.el in ~/.doom.d
(doom-load-envvars-file "~/.emacs.d/.local/env")
```

Figure 10.7
Environment configuration for doom-load-envvars-file.

After all these steps are successful reload configuration in Doom `[C-h r r]`. Finally, one can also utilize some features of Vim using evil[8] mode emulation in Doom.

10.4.8.2 Installing with the Recipes method Doom does not use the standard method of installing packages with the built-in package manager of Emacs. It uses another method using straight.el, a declarative package manager. If a package cannot be retrieved in this way, it allows the installation of a package from a source, for example hosted in GitHub or GitLab, from which one can declare the details of the project's files and also target a specific branch in a repository. The code in figure 10.8 must be added in the packages.el file in the Doom directory. The Package Management section in the Doom Emacs documentation provides a thorough explanation about best practices and examples on this topic.

```
(package! sclang-mode

:recipe (:host github :repo "supercollider/scel" :files   ("el/*.e
l")))
```

Figure 10.8
Installing scel package with the recipes method.

While the code is self-explanatory, it is worth mentioning that one may locate specific files, just in case these are placed in a sub folder of the repository.

10.4.9 Interactions with Other Modes in Emacs

The customization in figure 10.9 demonstrates how to interact with SC while working for example in another mode, such as TidalCycles, with the use-package macro.

```
;;interaction tidal-mode and sclang-mode
(use-package! tidal
:hook (tidal . sclang-mode)
:bind (:map tidal-mode-map
   ("C-c C-l" . sclang-start)
   ("C-M-z" . sclang-switch-to-workspace)
   ("C-c >" . sclang-show-post-buffer)
   ("C-M-x" . tidal-evaluate-region)
   ("C-c <" . sclang-clear-post-buffer)))
;;add personal key bindings in sclang-mode
(use-package! sclang-mode
:bind (:map sclang-mode-map
   ("C-c C-s" . sclang-server-reboot)
   ("C-c C-d" . insert-date-string))
```

Figure 10.9
Interaction with SuperCollider in Tidal Cycles.

Specifically, the example above shows how to run SuperCollider commands from a TidalCycles buffer, and thus be able to create interaction hooks between different languages. Manifestly, this may save some time, especially in contexts where SuperCollider and other systems are used simultaneously for live performances. It is more efficient to memorize several key bindings under one ecosystem, that is one single editor for all, than to use multiple platforms.

10.4.10 Some Tools Used for SuperCollider in Emacs Some useful extensions to integrate with SuperCollider are available in Emacs. These may be used for many tasks, from version control management, authoring technical documents and manuals, to plaintext files.

The author has found the following tools to be extremely useful to integrate with SuperCollider, both in order to boost productivity and to allow nifty project organization.

10.4.11 Org Mode

Org mode, advertised as "life in plain text," is a markup language for authoring academic papers and plaintext files and project management and for setting tasks and deadlines. Once an org-mode file is created (or Org mode is activated in a buffer) using the.org extension, it allows for creating markup elements, such as a title hierarchy, embedding links in text, table generation, calculation of formulas, and inserting and formatting images inside documents. Moreover, Org mode is a useful resource for literate programming and reproducible research. It may export files in an array of available formats, including plaintext, HTML, and Markdown, etc., and thus is a very convenient tool to generate resources that may accompany large projects, such as README files and manuals. Emacs supports Org editing, and several alternative configurations exist to make it look sleeker than the vanilla version.

10.4.12 Babel in Org Mode

Babel, supported in Org, allows for running a variety of programming languages at the same time, using active block code structures in plaintext documents. This is extremely convenient for teaching, presenting, or even keeping notes of experimental coding with SuperCollider. It allows one to work interactively with code snippets, writing descriptions for each bit of code, while at the same time integrating SuperCollider and its post window under the same hood.

Other languages can be run at the same time, allowing a focus on the content of a presentation and technical issues of the code at hand. The code is run with `[C-c]`, and results appear in either the post window or at the end of the block, depending on the language. An example of this is illustrated in figure 10.10.

```
#+TITLE: OrgMode Babel SC Demo
** Examples of SuperCollider & Org
This is an example of interacting Org and SC using Babel.
SC snippets are active blocks of code which can run with   results
 appearing in post window and also rendered in the   same document.
** Active SC Code blocks SC can be executed from inside Org   mode
 buffers using Babel. Active code blocks are part of the   document
which user may execute and render results at the   end. These block
s can be exported as source code in and   saved as SC document.
*** Examples of various active blocks
#+BEGIN_SRC sclang :var foo=0.65 :tangle yes :hlines yes
(0.5 * foo.reciprocal).postln;
"Babel and Emacs for SuperCollider".postln;
#+END_SRC

#+begin_src js :var a="hello" :tangle no
console.log(a)
#+end_src
#+RESULTS:
: hello
: undefined

#+BEGIN_SRC python :var a="Hello"
print(a)
#+END_SRC
#+RESULTS:
: None

#+BEGIN_SRC emacs-lisp
(setq sentence "Emacs loves SuperCollider.")
#+END_SRC
#+RESULTS:
: Emacs loves SuperCollider.
```

Figure 10.10
Active code block interaction in Babel and Org mode.

Arguments can be also passed in the blocks, such as a global variable (as seen in the SuperCollider example block). Tangle is a code block header, which may be declared in case you want to extract the code to a separate document. It is hard to explain thoroughly the package's attributes at length, but it has proven to be very handy when

working with SuperCollider code and other languages. To activate sclang support for Babel, install the *ob-sclang* extension found at this link.[9]

10.4.13 Magit: Version Control

A common task while working in larger SC projects is the use of version control using GitHub. This may be supported by windows-based interfaces or command-line software. However, Magit (https://magit.vc/) allows the integration of git commands in Emacs, such as push, pull, and checking status from inside a project. Magit is initialized in Doom via `[C-x g]`, which opens a new buffer. More information about the package and details for Github and Doom/Emacs integration can be found at https://docs.doomemacs.org/latest/modules/tools/magit/#/usage.

10.5 Conclusions

A number of available IDEs exist for editing SuperCollider code, integrating with other useful extensions, to help build a more personal ecosystem for SuperCollider. Of these, Visual Studio Code is the most straightforward and requires very few steps to establish communication with SuperCollider. Other editors such as Emacs allow for a greater level of customization, selecting from a plethora of available packages to enhance the coding process supporting many modes, including other editor emulations, such as Vim.

In the beginning, it may feel daunting and require some ability to comprehend Emacs Lisp in order to take full advantage of the system's capabilities and its installed modules. These may be installed in various ways, including via configuration frameworks such as Doom and Spacemacs. In addition to SuperCollider code editing, other environments and useful resources such as Babel allow code execution within plaintext files. In this context, the real-time correspondence between SuperCollider active code blocks and texts in the same document can be a very useful asset for authoring technical documents, presentations, and other type of output. Finally, while Emacs can support running multiple modes at the same time, it can also provide a uniform workflow during coding with SuperCollider and other languages.

Notes

1. https://survey.stackoverflow.co/2023/#section-most-popular-technologies-integrated-development-environment.
2. https://scsynth.org/t/which-ide-is-the-most-popular-for-supercollider-editing-code-nowadays/8257.
3. https://microsoft.github.io/language-server-protocol/.
4. https://github.com/magnars/multiple-cursors.el.
5. https://github.com/hlissner/doom-emacs.
6. https://www.spacemacs.org.
7. https://github.com/bbatsov/prelude.
8. https://www.emacswiki.org/emacs/Evil.
9. https://github.com/djiamnot/ob-sclang.

References

Kleene, R., 2020. “The Era of Visual Studio Code.” Accessed September 23, 2023, from https://blog.robenkleene.com/2020/09/21/the-era-of-visual-studio-code/.

McLean, A. 2014. “Making Programming Languages to Dance to: Live Coding with Tidal.” In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design,* pp. 63–70.

Petersen, 2015. “Mastering Emacs.” Accessed September 23, 2023, from https://books.google.com.tr/books?id=Gu7qsgEACAAJ.

Wilson, D., 2021. “M-x Forever: Why Emacs Will Outlast Text Editor Trends.” Accessed September 23, 2023, from https://emacsconf.org/2021/.

11 SuperCollider on Small Computers

Matthew John Yee-King

11.1 Introduction

This chapter explores the use of SuperCollider on a range of small computers, defined as cheap, single-board computers with ARM CPUs. Several exemplar systems have been selected: the NanoPi Neo, Beaglebone, and Raspberry Pi model 2 and 4. Different ways to pass audio into and out of the computers are considered, including direct wiring, daughterboards, and USB sound cards. Audio quality measurements for these I/O (input/output) methods are examined, using total harmonic distortion + noise and signal-to-noise ratio metrics. The SuperCollider performance capabilities of these exemplar systems are measured through the number of oscillators and FFTs they can run under various conditions. The chapter concludes with several case study projects which demonstrate the kinds of applications that one might consider for SuperCollider on small computers.

The first computer I used to run SuperCollider was an Apple PowerBook G4 with macOS 9.2 in the year 2000. Back then, Apple Macs and macOS were the only computers and operating systems capable of running SuperCollider. I remember looking at the muscular, beige-box Intel PC with its twin Pentium III CPUs next to my titanium-encased Mac and wishing it could run SuperCollider too!

Things soon changed. In 2002, James McCartney open-sourced SuperCollider, leading to it being ported to GNU/Linux and Windows. At the same time, rapid changes were happening to the availability and power of small computers. In 2005, the microcontroller-powered Arduino platform appeared, leading to an explosion of DIY electronics and music projects (Richards, 2017; Edstrom, 2016). In 2007, Apple launched the ARM CPU-powered iPhone, and 2008 saw the first Android phone with its Linux kernel and ARM CPU. Musicians were quick to experiment with the audio capabilities of these new platforms, for example, with the Stanford Mobile Phone Orchestra founded in 2007 (Wang et al., 2014) and Dan Stowell's experimental port of SuperCollider to Android (and, therefore, ARM) in 2010. In 2012, the Raspberry Pi was released. Unlike Android and iOS-based smartphone devices, the Raspberry Pi is very cheap and can be easily integrated with DIY electronics. Unlike Arduino, the Pi

can hold a full GNU/Linux operating system, enabling it to run SuperCollider. Since 2012, a series of more powerful Pis have been released, as have many other ARM-based, Linux-running, cheap, and small computers. Running SuperCollider on these small computers is the subject of this chapter.

For our purposes, we will limit "small computer" to mean a single-board computer (SBC) running the Linux kernel on an ARM CPU architecture. Henceforth, I shall refer to these as "ARM SBCs." Other physically small computers are available, such as micro-itx format Intel CPU machines, Intel NUCs, and Mac Minis. We will not consider these as they are just physically smaller versions of PCs or Macs, and they miss several of the advantages and features of ARM SBCs discussed below.[1] The Arduino and other microcontroller boards cannot directly run SuperCollider, so we will not consider them in this chapter. (See chapter 4 for further discussion of using an Arduino.)

11.1.1 Examples of Small Computers

There are many ARM SBCs available, and Ojo et al. (2018) present a comprehensive review of these. Perhaps the best-known ARM SBC is the Raspberry Pi. so, we will consider two models of the Raspberry Pi in the following sections. We will also look at two other systems: the FriendlyElec NanoPi Neo[2] and the Beaglebone Black.[3] Each of these systems has some unique characteristics that differentiate it from the others. I will describe those characteristics in section 11.2. These systems also have different methods for passing audio into and out of SuperCollider, and this results in differing audio quality. I will cover audio I/O options in section 11.2 and audio quality in section 11.4. The four systems differ in their CPU and RAM specifications, leading to variations in DSP performance. I will present an analysis of this performance in section 11.5.

There are other ARM SBCs that could potentially run SuperCollider. Examples are the other NanoPi models, the Rock Pi 4, the Odroid 2, the Khadas VIM series, and the NVIDIA Jetson series. These less common ARM SBCs differ in the amount of RAM, hard drive support, number of HDMI outputs, USB ports, etc. They also differ in their multicore CPU architectures. For example, some have big/small setups which combine high-power, high-energy cores with low-power, low-energy cores. If you are using the single-core, vanilla version of scsynth as recommended by the developers,[4] the ARM CPU multicore design is unlikely to make much difference to the straight-ahead DSP performance between two equivalently clocked ARM CPUs.

Another difference between the systems we test here and some of the other ARM SBCs is the presence of a neural engine chip. Neural engines can run pretrained neural network models to perform tasks such as object classification in video streams (Süzen et al., 2020). SuperCollider cannot use neural engines directly, but you can interact with them via OSC messages. For example, you could write a Python script that talks to the

neural engine and then send the outputs to SuperCollider via OSC messages. This idea is beyond the scope of this chapter, but it is certainly something to consider if you are investigating the best ARM SBC for your needs.

Taking all this into account, the set of systems covered here provides a decent enough range of capability and features. The readers should be able to extrapolate from the performance and audio I/O evaluations presented in sections 11.4 and 11.5 to what they can expect from faster boards.

11.1.2 Reasons for Using ARM SBCs to Run SuperCollider

ARM SBCs can be cheap, quiet, small, energy-efficient, and surprisingly performant; they can provide professional-quality audio I/O and allow you to connect homemade electronics and sensors. Let us briefly consider those characteristics in turn.

The first is cheapness: a Raspberry Pi costs between $15 and $50, which is 1/10th or less of the price of an all-purpose laptop, even including a power supply, case, and audio adapter. The cheapest system that I evaluate here is the FriendlyElec NanoPi Neo, and it is listed at $16 on the manufacturer's website.[5] Being cheap opens up the possibility of a unit performing a fixed role or multiple units working together. (There will be more on this in section 11.6, which presents some case studies.)

The second motivator for ARM SBCs is smallness: a typical ARM SBC fits in a case that is approximately 10 x 5 x 5 cm. The NanoPi Neo board is a diminutive 4 x 4 cm. This is a lot smaller than the twin Pentium III PC I mentioned in the introduction to this chapter. Being small means you can put multiple devices in your bag when going out to a gig, just as you might typically do with minisynthesizers and effects pedals. You can also easily conceal them in a gallery installation setting.

The third element is quietness: some ARM SBCs are passively cooled. If not, they use a small fan. This means that they might be preferable to an actively cooled computer for use in a studio environment.

The fourth is performance. This characteristic is considered in more detail in section 11.5, but the executive summary at this point is that you can run many concurrent SuperCollider synths, FFTs, etc., on an ARM SBC. We will cover audio I/O quality in section 11.4, but to summarize here, ARM SBCs can use their built-in audio I/O, USB audio adapters, or high-quality audio daughterboards to provide excellent-quality I/O.

Finally, ARM SBCs offer many options for integrating with sensors and homemade electronics. This might not help you through the security check at the airport, but it does mean that ARM SBCs are great platforms for DIY projects such as synthesizers with unique controllers and so forth. We will consider some specific use cases in section 11.6.

11.2 The Small Computers Selected for Analysis

In this section, you will find a description of the ARM SBCs selected for in-depth testing in later sections. Four systems are tested: the FriendlyElec NanoPi Neo, the Beaglebone Black, the Raspberry Pi 2 Model B v1.1, and the Raspberry Pi 4 Model B 4GB. Table 11.1 summarizes the features of the systems tested, and figure 11.1 contains images of the boards.

Table 11.1
Overview of features for tested ARM SBCs

Note: Recommended power is the output of the power supply for that system recommended by the manufacturer; 100 sine power is the power usage reported by the powertop application when running 100 sine synths in SuperCollider.

Board	NanoPi Neo	Beaglebone Black	Raspberry Pi 2	Raspberry Pi 4
Cost	$16	$40	$25	$50
CPU clock (GHz)	1.2	1	1	1.5
CPU cores	4	1	4	4
CPU version	ARM7	ARM8	ARM7	ARM8
RAM (GB)	0.5	0.5	0.5	4
Wi-Fi	No	No	No	Yes
Ethernet	Yes	Yes	Yes	Yes
HDMI	No	1 (not with Bela)	1	2
Add-on audio	Not tested	Bela hat: stereo I/O	Not tested	HifiBerry ADC+DAC Pro
Built-in audio	2 in, 2 out	No	2 out	2 out
Recommended power	10 W	5 W	10 W	12.5 W
100 sine power	Unknown	Unknown	2 W	3 W

11.2.1 The NanoPi Neo

FriendlyElec released the NanoPi Neo in 2016 (see figure 11.1). The Neo has an Allwinner H3 Quad-core Cortex-A7 CPU running at 1.2 GHz and 512 MB of DDR3 RAM. The Neo expects a power supply unit (PSU) capable of delivering 2A at 5V, though it does not require that much current constantly. The Neo has a GPIO header, which supports the i2s digital audio protocol, and an S/PDIF digital output connector, though these were not tested. The Neo also has stereo analog audio out and stereo audio in pins which allow for the connection of standard audio jack ports, as shown in figure 11.2.

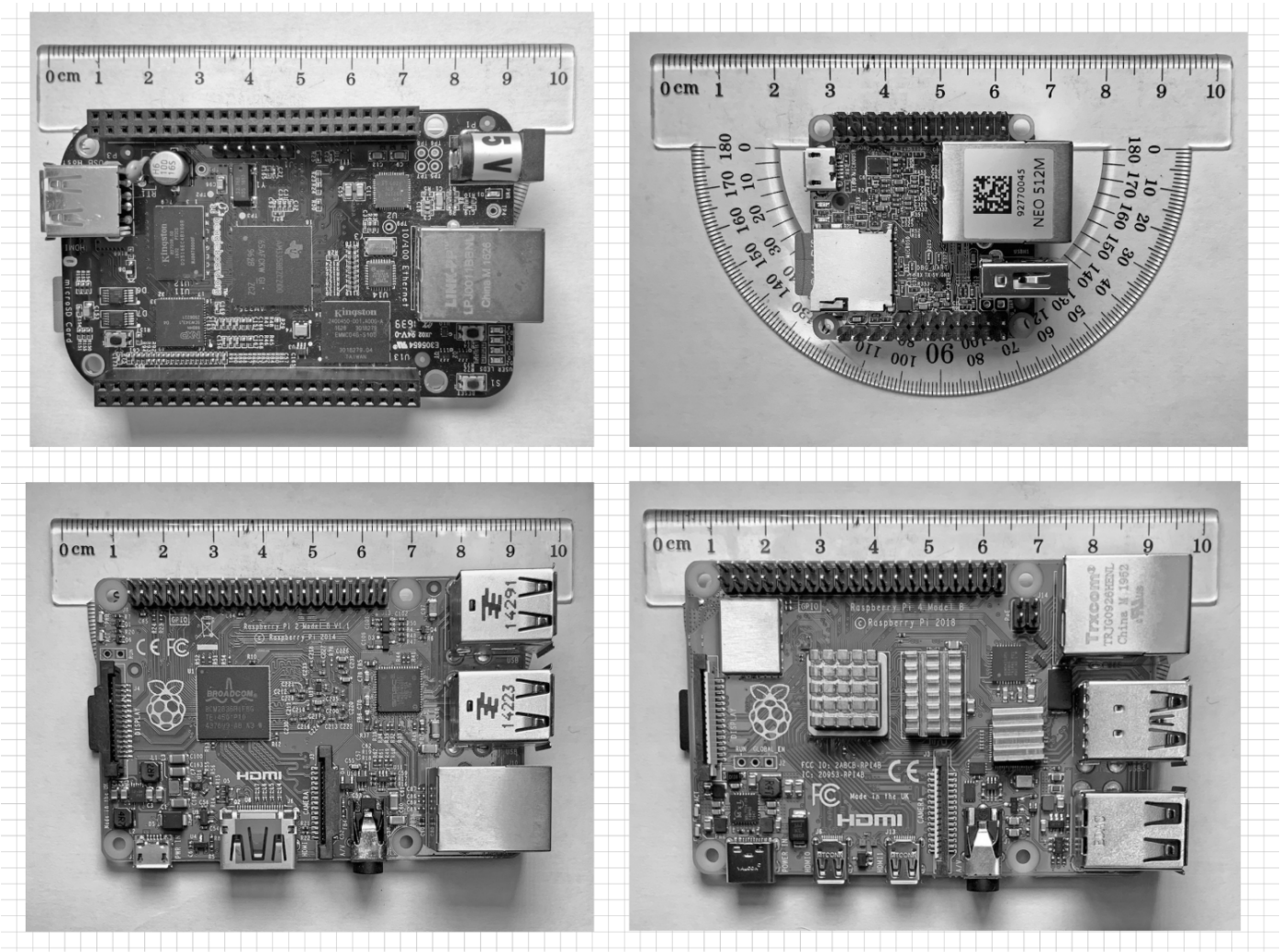


Figure 11.1
The four boards tested. Clockwise from top left: Beaglebone Black, NanoPi Neo, Raspberry Pi 2 B v1.1, and Raspberry Pi 4 B.

The audio I/O appears as an ALSA device under the Linux OS images provided by FriendlyElec; The Linux audio server jackd can access this device, and therefore, so can SuperCollider. (There will be more on jackd later in this discussion.) The device supports 16-bit audio (not 24-bit) at sample rates of 44,100 and 48,000 Hz and can reliably use a block size as low as 64 samples. The wiring for the NanoPi audio output is shown in figure 11.2. I will present a more detailed analysis of this audio system in terms of its sound quality and DSP performance in sections 11.4 and 11.5. An audio daughterboard is available for the Neo; it provides an external digital to analog converter and communicates with the mainboard via i2s. Since the built-in system supports input and output and the daughterboard was not available, I only tested built-in and USB audio.

11.2.2 The Beaglebone Black

The Beaglebone Black (BBB) shown in figure 11.1 was released in 2013. A miniature version of the BBB was released in 2017 and Beagleboards with stronger CPU and AI capabilities were released in 2016 and 2019, but the BBB is compatible with the Bela daughterboard discussed below and that is why I have selected it for testing.

The BBB has a single-core Cortex-A8 CPU running at 1 GHz with 512 MB of DDR3 RAM. It has a lower power requirement than the other SBCs of 5 W, so it will reliably power from a USB port on a computer. It can network over the same USB connection with a computer, and this is a useful development setup. Another differentiating feature of the BBB is its eMMC memory, which makes it possible to boot the OS directly from the board instead of from a separate memory card. In addition to the main CPU, the BBB has two 200-MHz programmable real-time units (PRUs). The PRUs provide high-bandwidth, low-latency transfer between I/O pins and system memory independent from the CPU. The PRUs are the main difference between the Beaglebone and the other ARM SBCs here as regards SuperCollider, as we will explain below when we discuss the Bela add-on board. The BBB does not have any built-in analog audio I/O, but it can send audio over the HDMI connector, and there are daughterboards (capes) that provide analog audio I/O.

11.2.3 The Bela Cape

The Bela is a daughterboard (cape) for the BBB. It is also available in a cheaper, miniaturized form for the Pocket Beaglebone, which is similarly specified to the BBB tested here. The hardware, shown in figure 11.2 with wiring features stereo audio I/O as well as 8 channel analog and digital I/O. The Bela software runs the Linux kernel as a service alongside the audio and other I/O systems, and it offloads raw I/O processing to the PRUs. This is a step beyond the real-time audio setup that Linux audio users might be familiar with because the audio process is separate from and has equal or greater priority than the Linux kernel. This arrangement permits very low latency and consistent audio performance. To allow for this technical setup, Bela has a custom audio subsystem instead of the ALSA and jackd combination used by the other boards. Bela's developers provide a special SuperCollider server driver which communicates with Bela instead of jackd. The developers also provide SuperCollider UGens to communicate with the analog and digital I/O but here we will focus on the audio I/O capabilities of the Bela.

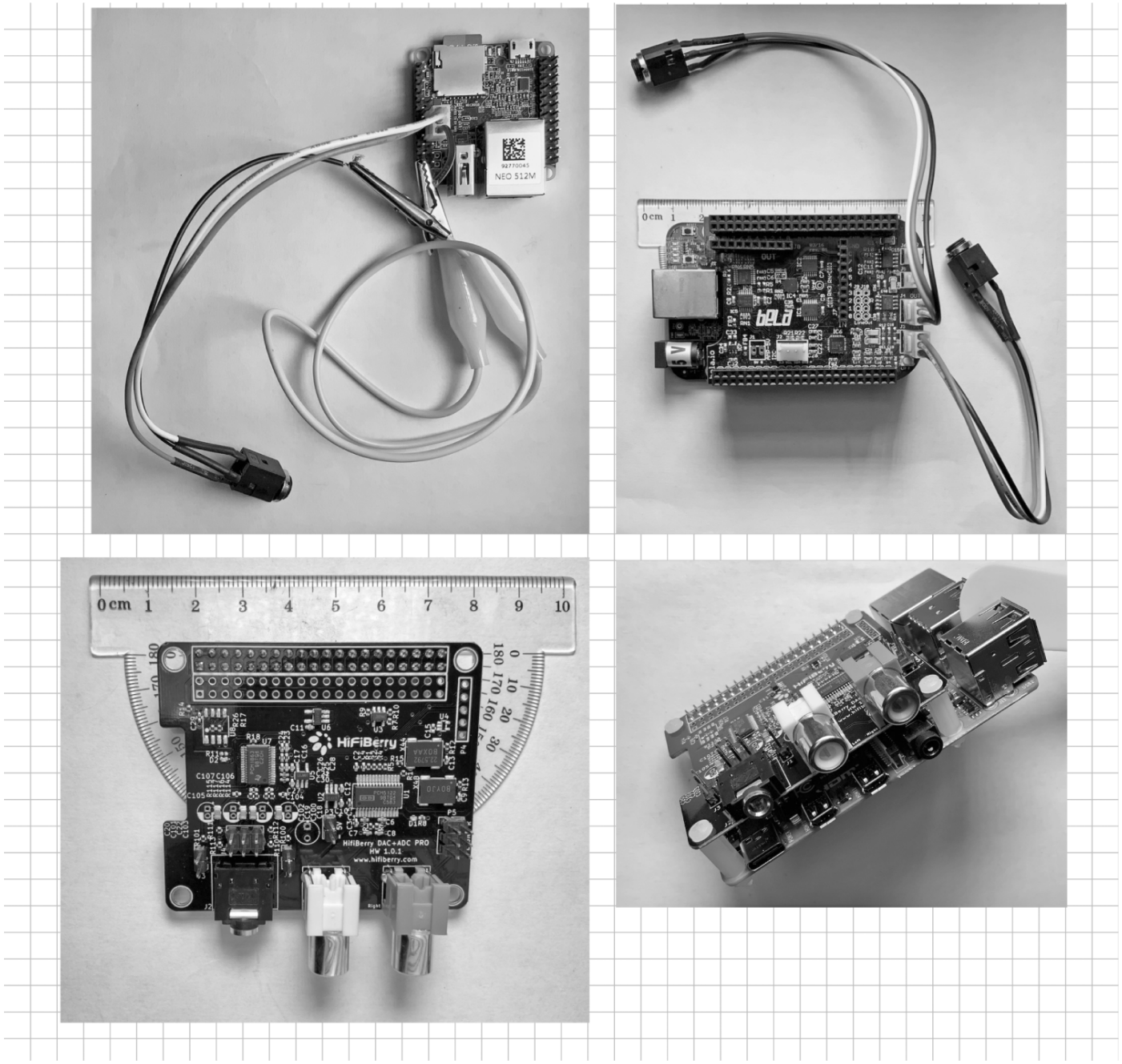


Figure 11.2
The ARM SBCs with audio connections and add-ons. Clockwise from top left: NanoPi Neo, with wiring that might offend hi-fi enthusiasts despite impressive audio quality metrics; Bela, mounted on Beaglebone with connectors; HiFiBerry DAC+ ADC Pro board; and HiFiBerry mounted on Pi 4.

11.2.4 The Raspberry Pi

The Raspberry Pi was launched in 2012 and has since seen multiple iterations. The systems tested here are the Raspberry Pi 2 Model B v1.1 (2015) and the Raspberry Pi 4 Model B (2018). They are shown in figure 11.1. These two systems are pretty representative of the range of capabilities of Pis, especially since the Pi 2 shares a CPU with some Pi Zero models. The Pi 2 and Pi 4 have quad-core ARM7 and ARM8 CPUs

running at 1 GHz and 1.5 GHz. They have 512 MB and 4 GB of RAM and boot from micro-SD cards. They both include a stereo analog audio output jack, and I test these with SuperCollider later in this chapter. Users can enhance the audio and other capabilities of the Pi using daughterboards called "hats." Many hats provide stereo audio output via what their manufacturers describe as "high-end digital to analog converters." A few hats also provide audio input. Here we will test one such hat—the HiFiBerry DAC+ ADC Pro. This hat provides stereo I/O at sample rates up to 192KHz and is shown in figure 11.2.

I would also like to mention the Norns system from Monome (Marasco, 2022), which provides a software API for developing audio applications and an audio I/O hat for the Raspberry Pi. Norns uses SuperCollider as its underlying synthesis engine. The audio hat includes physical controls such as knobs and buttons and a small display, so it is a neatly integrated system for running SuperCollider on Raspberry Pi. Norns comes as a complete unit with an integrated Raspberry Pi or a DIY hat version. Norns is quite expensive compared to the other options here—the complete unit is $800, and the kit is $375.[6] I did not test Norns.

11.3 Setting Up an ARM SBC

Now that we have explored the four main systems, we shall consider how to set them up to run SuperCollider. If you are experienced in Linux audio, you will be familiar with some of the concepts, but there are still useful tips here for you. The basic requirements to run SuperCollider on an ARM SBC are an operating system, the jackd software, and the SuperCollider software. The SuperCollider stack is shown in figure 11.3.

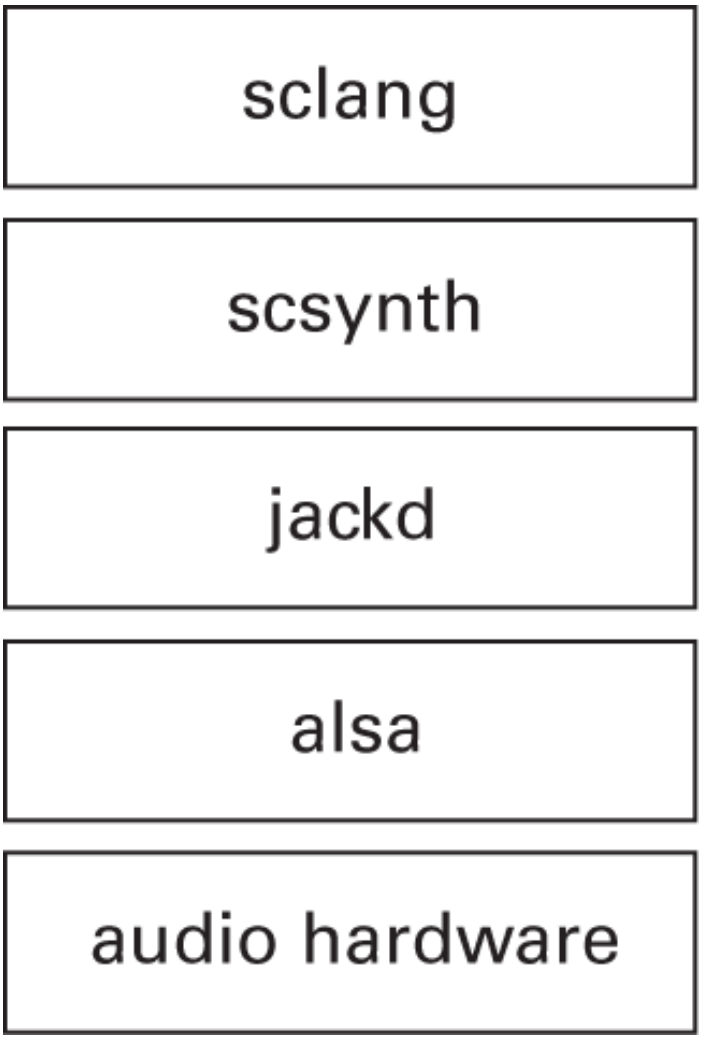


Figure 11.3
The SuperCollider stack. sclang talks to scsynth, scsynth talks to jackd (and sclang), jackd talks to ALSA, ALSA talks to the audio hardware. Bela replaces jackd and ALSA with a custom audio driver setup.

11.3.1 Installing the Operating System

I installed a GNU/Linux OS on each of the systems according to instructions available online. There was some variety in how this was done, but the first step was always to locate an appropriate ISO file containing the files for the OS and to image it onto an SD card. For the NanoPi, FriendlyElec provides an Ubuntu Core image. The Raspberry Pi runs a variant of Debian, as does the Beaglebone. To run with the Bela cape, the Beaglebone uses a custom image provided by the makers of Bela.

The Beaglebone was the most complex install, as it defaults to booting from the onboard eMMC storage, and it is normally necessary to press a button on the board to make it boot from the SD card. Unfortunately, the button does not work when the Bela is in place, so booting into the Bela OS required deliberately corrupting the eMMC to force the SD card boot or connecting some wires to the board in a certain configuration. Once booted, it is possible to flash the SD card OS to the eMMC.

Once you have the OS set up, you can connect the machine to a screen using an HDMI cable and log in with the default user/password combination. If the machine does not have HDMI like the NanoPi (or the BBB once the Bela is installed), or you just want to access it remotely, you can use a terminal program running on another machine to SSH into the ARM SBC with the default user and password. The easiest way to manage this is to connect your machine and the ARM SBC to a router using Ethernet cables. Most OSs will use DHCP to automatically request an address on the network, and a quick look at your router's control panel will tell you what is connected and what its address is. The Raspberry Pi OS that I used had SSH disabled by default, so I had to first connect it to a screen, log in, and then run the raspi-config tool on the command line to enable SSH.

The BBB provides another networking option, which is to connect it to a USB port on your main machine, whereupon it creates a network over USB with your machine and itself. You need to install drivers to make this work on Windows and Mac, but it worked automatically on my Linux machine. I could then SSH directly to the connected Beaglebone at its default address of 192.168.7.2.

11.3.2 Jackd

Once you have the operating system installed and you have established an SSH connection, the next tool that you need is jackd, a powerful software tool designed for pro-audio use which provides a consistent interface between audio applications and audio hardware. So long as jackd can talk to your audio hardware, any jackd-compatible application (like SuperCollider's scsynth) should be able to talk to jackd. So an essential property of any ARM SBC audio hardware that you want to use for SuperCollider is jackd compatibility. One exception is that jackd is not necessary with Bela, as it uses a different means to communicate with the audio hardware. In fact, if you are using Bela, you can skip the rest of this section.

Under Linux, jackd compatibility normally requires an ALSA driver. Jackd talks to ALSA, and ALSA talks to the hardware. You can see a list of available ALSA devices on your Linux system with the following command:

```
aplay -l
```

You should see various devices listed. Figure 11.4 shows an example of what you might see on a Raspberry Pi 4 with a HifiBerry hat fitted. Additionally, if you wish to use a USB audio adapter and it is class-compliant, it should automatically appear in your aplay list once plugged in. In the tests presented here, I used an RME Fireface UCX II as a reference sound card. The RME has a class-compliant mode that allows ARM SBCs to use it.

```
pi@raspberrypi:~ $ aplay —l
**** List of PLAYBACK Hardware Devices ****
card 0: Headphones [bcm2835 Headphones], device 0: bcm2835   Headph
ones [bcm2835 Headphones]
   Subdevices: 7/8
   Subdevice #0: subdevice #0
   Subdevice #1: subdevice #1
   Subdevice #2: subdevice #2
   Subdevice #3: subdevice #3
   Subdevice #4: subdevice #4
   Subdevice #5: subdevice #5
   Subdevice #6: subdevice #6
   Subdevice #7: subdevice #7
card 1: vc4hdmi0 [vc4-hdmi-0], device 0: MAI PCM i2s-hifi-0   [MAI
 PCM i2s-hifi-0]
   Subdevices: 1/1
   Subdevice #0: subdevice #0
card 2: vc4hdmi1 [vc4-hdmi-1], device 0: MAI PCM i2s-hifi-0   [MAI
 PCM i2s-hifi-0]
   Subdevices: 1/1
   Subdevice #0: subdevice #0
card 3: sndrpihifiberry [snd_rpi_hifiberry_dacplusadcpro],   device
0: HiFiBerry DAC+ADC Pro HiFi multicodec-0 [HiFiBerry   DAC+ADC Pro
HiFi multicodec—0]
   Subdevices: 1/1
   Subdevice #0: subdevice #0
```

Figure 11.4
The output of aplay -l on a Raspberry Pi 4 with a HifiBerry hat attached. Card 0 is the built-in audio, cards 1 and 2 are the double HDMI ports, and card 3 is the HifiBerry. The card number is important, as that is the device number to pass to jackd.

Also, jackd permits arbitrary virtual wiring between jackd-compatible audio applications and the audio hardware. For example, you can wire SuperCollider directly to your DAW to send and receive audio, assuming that your DAW is running on the same machine, or you can share the audio hardware between multiple applications.

Note that jackd can be a little tricky to use, as you have to specify low-level audio settings such as the target device, sample rate, bit rate, and so on. These concepts should be familiar to anyone who has configured an audio device in a DAW such as Logic, Reaper, or Cubase. The difference is that here we will configure jackd on the command line using flags. There is a graphical application to configure jackd called qjackctl, but working on the command line is a typical interaction mode for ARM SBCs.

You can install jackd with the package manager apt using the following command:

```
sudo apt install jackd
```

The package manager will want to install quite a few different components, but they only use space; they should not affect performance. Once the install process completes, you will have a new program available on the command line called jackd. Here is an example command, which starts jackd on a Raspberry Pi's internal audio device to send audio out of the mini-jack socket:

```
jackd -R -d alsa -r 44100 -P hw:0 -o 2 -S
```

`-R` runs jackd in real time, meaning that it has a higher priority on the system, `-d` also uses the ALSA driver, `-r 44100` sets the sample rate to 44100, `-P hw:0` sets the playback device to the first device listed by aplay, `-o 2` sets the number of channels to 2, and `-S` tells it to use 16-bit samples. Other audio setups might have audio inputs and support different sample rates and bit rates, in which case you can run something like this:

```
jackd -R -d alsa -r 48000 -d hw:3
```

Here, I used a second `-d` option in place of `-P`. This means that it will create inputs and outputs, not just outputs. I did not specify the bit rate or the number of outputs, so it will probe the sound card and try to find a working option. This does not always work, which is why I specified it in the first jackd command above. Put an ampersand `&` at the end of that command, and it will run in the background and return you to a prompt where you can run further commands. You can stop the jackd process later with this command:

```
killall jackd
```

11.3.3 Installing SuperCollider

Once you have jackd installed, you are ready for the SuperCollider setup. The Bela OS image comes with a special Bela compatible version of SuperCollider preinstalled, so there is no need to follow these instructions for Bela. Otherwise, the simplest option is to install SuperCollider using prebuilt packages available via the apt package manager. The following command should install everything:

```
apt get install supercollider
```

This will provide you with the interpreter program sclang, the server program scsynth, and the IDE scide. With these, you can use SuperCollider via the graphical IDE just as you would on any other platform. The problem is that this build assumes you are logged on to the machine and have a display connected to it. Without a physical display, you cannot run the IDE or sclang on the command line.[7]

The solution to this problem is to use a custom GUI-less build. If you wish to run sclang remotely via SSH (a protocol used for remote shell access), or if you want your board to start up and run some sclang commands without any interaction, you need a GUI-less build. There are detailed and up-to-date instructions on how to do this on the SuperCollider GitHub page,[8] so I refer the reader there. Essentially, you have to install some development packages using the apt tool and then use CMake to build the SuperCollider programs with some special flags to disable the IDE and other features of the build. The GUI-less build instructions tell you to compile jackd yourself, but I found it was easier to install jackd with the apt tool, even though it also installed other components. I could still achieve what I wanted, which was to run jackd and sclang on the command line via an SSH session, and I avoided the slightly tricky jackd build. The SuperCollider build takes quite a long time, so it is a good opportunity to go for a walk while it is running.

11.3.4 Starting SuperCollider

Once you have built and installed SuperCollider (and jackd), you are ready to test your installation. Assuming you have successfully started jackd, the next command to run is

```
sclang
```

Then you should see the sclang prompt, whereupon you can ask it to play a sound:

```
s.waitForBoot{
{SinOsc.ar(440, mul:0.25).dup}.play
};
```

You should hear a sine tone coming out of both channels. If you cannot hear it, quit sclang and run the following command to configure the mixer of your audio device:

```
alsamixer -d 0
```

Set `-d` to the card you want from the `aplay list`. Figure 11.5 shows alsamixer pointed at the HifiBerry. It exposes all the controls that the low-level driver provides,

so it can be quite confusing. You can interact with alsamixer with the cursor keys to select and alter channel levels, and the `m` key to switch channels on and off.

Figure 11.5
The command line mixer alsamixer, controlling a HifiBerry hat.

11.3.5 Run SuperCollider at Boot

For some of the case studies presented in section 11.6, you will want to configure jackd and SuperCollider so they start automatically upon booting. For example, the ARM SBC can boot and run an effect processor-type synth on the input, or it might just start the SuperCollider server with some particular settings and sit on your network waiting for OSC messages. To achieve this, I refer the user to the excellent README files on the SuperCollider GitHub page,[9] as the exact instructions are likely to change over time. The basic principle is that you can set up the ARM SBC to run commands on boot. One of those commands is to start jackd; another runs sclang with a script that starts your scsynth server.

11.4 Audio I/O Quality

In this section, you will find an analysis of the audio quality achieved using the various audio I/O options mentioned in section 11.2. The test asks two questions: How accurate is the signal reproduction, and how clean is the signal path? I chose to measure two standard characteristics of audio systems: total harmonic distortion + noise (THD+N) and signal-to-noise ratio (SNR). THD+N tells you if any additional harmonics are

added to a test tone by the audio system, and SNR tells you how much background noise the audio system creates. To be more specific, THD+N is the summed power of the harmonics in a signal once you have detected and removed the first harmonic using a notch filter. The test signal is, by convention, a 997-Hz sine tone, so if it is accurately reproduced by the audio system, there should not be any power in the upper harmonics (1994 Hz, 2991 Hz, etc.).

11.4.1 Test Method

I set up each ARM SBC audio configuration in turn and recorded its output into the Reaper DAW software using a reference-grade Apogee Symphony Desktop sound card running at a 192-KHz sample rate with 24-bit samples. The manufacturer of the Apogee device lists its analog-to-digital THD+N as −113 dB, which means it should add absolutely minimal distortion of its own to the signal. I ran all ARM SBCs at a 44,100-Hz sample rate, as this was the only sample rate supported by Bela. Some ran at 24 bits, and some ran at 16 bits because those were the available bit depths.

When testing the inputs of the devices, I generated the 997-Hz tone out of the Apogee at a 192-KHz sample rate and recorded it into the ARM SBCs. For the Raspberry Pi and NanoPi input tests, I used `arecord` on the command line to record. I could have recorded into SuperCollider, but arecord was the simplest and most direct way to capture the signal. For the Bela, I used a C++ program that came as an example with the OS install because ALSA (and therefore `arecord`) are not part of the audio stack on Bela. To calculate the THD+N, I generated the signal using the following SuperCollider code:

```
{SinOsc.ar(997)}.play;
```

I calculated the THD+N value from the WAV file recording using Endolith's waveform analysis Python package.[10] I compared the results to a digital test tone, which was a sine tone digitally recorded directly from SuperCollider using its record function.

To calculate the output SNR, I recorded the output of SuperCollider when it was not running any synths and measured the average RMS level of the signal, where the RMS was calculated over frames of 2,048 samples. I used the librosa library's librosa.feature.RMS function to calculate the RMS (McFee et al., 2015). It was convenient to use librosa, as opposed to SuperCollider, for the analysis since the THD+N feature was also implemented in Python. To record the input SNR, I recorded on the device using `arecord` or the Bela C++ program with the silent Apogee connected to the input.

In summary, there were four tests:

1. Output: Generate a 997-Hz sine tone on ARM SBC @ 44,100-Hz best available bit rate, record into Apogee @ 192 KHz, 24 bits, and measure THD+N.
2. Output: Record "silence" from the ARM SBC @ 44,100-Hz best available bit rate into Apogee @ a 192-KHz, 24 bits, and measure the received signal level.
3. Input: Generate a 997-Hz sine tone with Apogee @ 192 KHz 24-bit and record into ARM SBC @ a 44,100-Hz best available bit rate.
4. Input: Record silent Apogee @ 192KHz 24-bit into ARM SBC @ 44100Hz best available bit rate.

11.4.2 Audio I/O Quality Results

Table 11.2 presents the results of the audio I/O quality tests. The RME referred to is the RME Fireface UCX-II, which I used as a benchmark. These data figures are probably not comparable to officially quoted figures in the manufacturers' literature, as I did not have access to calibrated test equipment, but they are suitable for comparison between the tested systems. For example, the RME manual has THD+N on the output at −104 dB and the HifiBerry datasheet states a THD+N figure on the output of −93 dB. My figures are 30 dB higher in both cases.

Table 11.2
Results of the audio I/O quality tests
Note that in this table, lower is better. The digital reference was directly recorded to WAV from SuperCollider, so it does not have any noise. The Raspberry Pis do not have a built-in audio input, so it is not possible to measure that.

Setup	Out THD+N dB	Out SNR dB	In THD+N dB	In SNR dB
Digital reference	−108	No noise	−108	No noise
RME (NanoPi)	−76	−87	−92	−128
Beaglebone Bela	−66	−77	−78	−58
NanoPi built-in	−73	−85	−29	−55
Pi 2 built-in	−34	−76	Not possible	Not possible
Pi 4 built-in	−54	−76	Not possible	Not possible
Pi 4 HifiBerry	−65	−97	−86	−109

Considering the audio outputs and THD+N, the Raspberry Pi 2's built-in audio system generated more harmonic distortion than the others. As I increased the gain in alsamixer toward 0 dB, clipping was audible. Most surprisingly for me was that, even with the amateur wiring shown in figure 11.2, the Neo performed very well on this test. The test correctly places the $1600 RME as the best-performing unit for THD+N, and it detects the negligible distortion on the digital reference tone. The HifiBerry beat the RME for output SNR, which is impressive.

Concerning the inputs, the NanoPi performed worst. I had some challenges setting the input level on this device, as there were many confusingly named elements in alsamixer,

which seemed to affect the incoming signal in strange ways. I am not convinced that I found the optimal settings for the NanoPi. The Bela performed well, and the HifiBerry very well. The RME achieved the best score.

11.5 SuperCollider Audio Performance

In this section, you will see a real-world SuperCollider DSP performance test. This test asks two questions: How many sines can the boards generate, and how many FFTs can they run without the audio stream glitching?

11.5.1 Test Method

The sine test involved manually running the following SuperCollider code with different jackd buffer sizes on the ARM SBC:

```
{300.do{arg i;
   i.postln;
   {SinOsc.ar(100, mul:0.0001)}.play;
   0.05.wait;
}}.fork;
```

As you can see, the code creates a lot of SuperCollider synths running a single sine oscillator plus associated output nodes generated by the `play` command. The code uses a fork to enable waiting between synth launches, and this made it easier to identify when the system had reached maximum load. The FFT test used the following code:

```
{20.do{arg i;
   i.postln;
   {var in, chain;
        in = WhiteNoise.ar(0.1);
        chain = FFT({LocalBuf(2048, 1)}, in);
        IFFT(chain)
   }.play;
   0.25.wait;
}}.fork;
```

This code makes synths that generate a white noise signal, convert it to the spectral domain, and then convert it back out to the time domain. This is a much more intensive task than the sine task.

To run these tests, I started jackd with a particular block size and then manually ran the code repeatedly, increasing the number of synths until I saw regular xruns. Xruns are caused when SuperCollider does not generate its output buffer quickly enough. They are

a sign that the computer cannot handle the load and are generally associated with audio glitching. I manually identified how many synths caused xruns to persistently happen as I found a simple script to detect the first xrun was not as reliable. I made sure I had set the system power governors to performance mode, which causes the CPU to sit at its maximum clock speed. The following command will do that:

```
echo performance | sudo tee \
/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

11.5.2 Audio Performance Results

Table 11.3 shows the best scores achieved in both tests for all systems tested. I have included somewhat remarkable scores from an M1 Mac Mini (technically an ARM SBC) and those from a ThinkPad X1 with an Intel i7 gen 10 CPU for reference. Figures 11.6 and 11.7 plot the scores across all buffer sizes for all systems.

Table 11.3
Audio performance test results

Setup	Lowest Possible Buffer	Sine Score (buffer size)	FFT Score (buffer size)
BBB Bela	8	500 (1,024)	35 (512)
NanoPi built-in	64	270 (512)	40 (512)
NanoPi RME	64	280 (1,024)	40 (512)
Rpi 2 built-in	512	320 (1,024)	45 (1,024)
Rpi 2 RME	64	310 (1,024)	40 (1,024)
Rpi 4 built-in	512	900 (512)	190 (1,024)
Rpi 4 RME	64	900 (1,024)	190 (1,024)
Rpi 4 HifiBerry	64	910 (1,024)	190 (1,024)
Mac Mini M1 RME	1	6000 (512)	1300 (512)
Intel i7 RME	64	2900 (1,024)	900 (1,024)

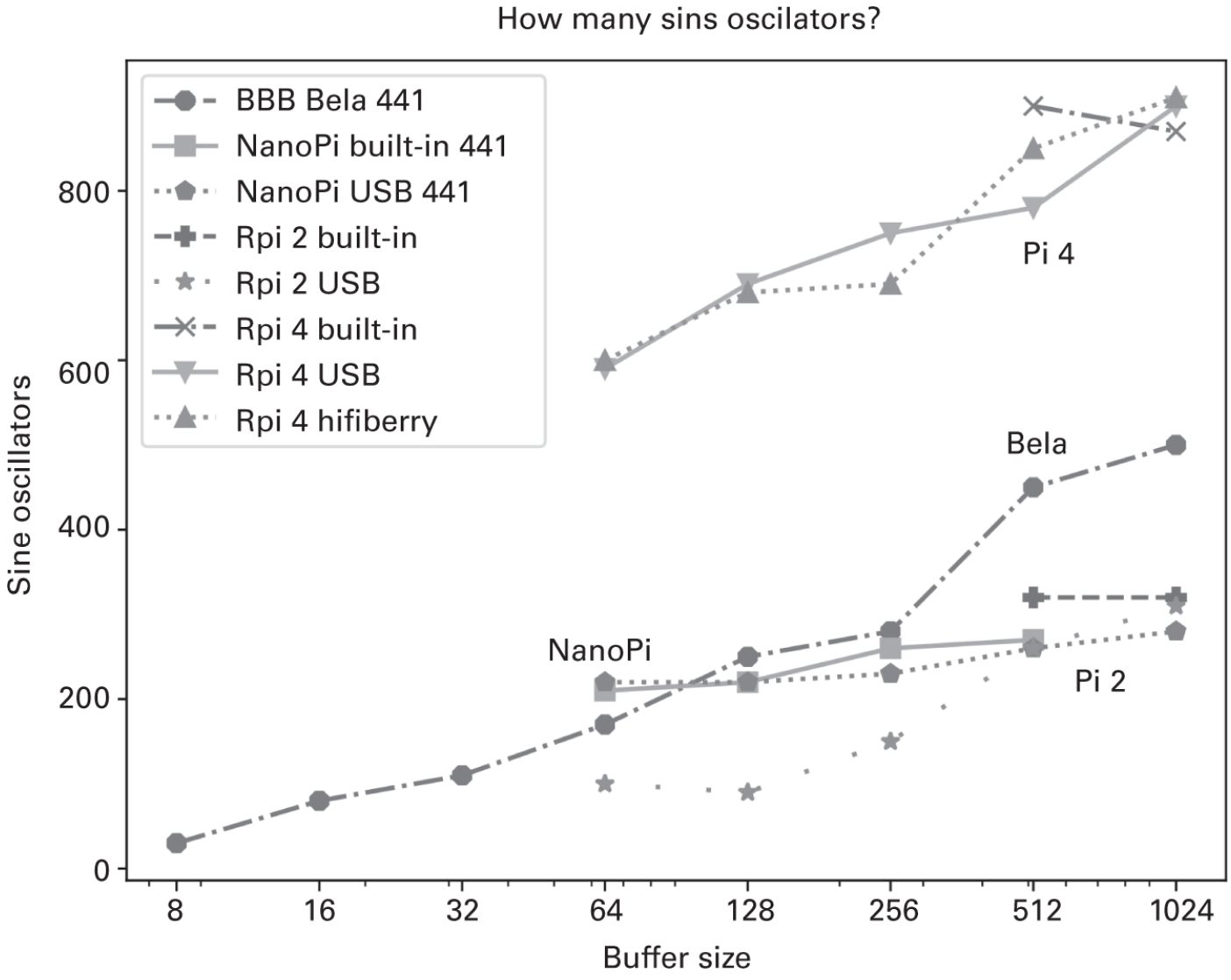


Figure 11.6
How many sine synths each ARM SBC can run at a sample rate of 44,100 Hz and various buffer sizes.

All tests were at a 44,100-Hz sample rate. The lowest possible buffer is the lowest buffer that that system supported. The Mac mini-RME tests used the manufacturer's proprietary driver. The Sine and FFT scores show the best score achieved of all possible buffer sizes and the buffer size used to gain that score.

The Raspberry Pi 4 has impressive performance, which is sufficient for a wide range of typical SuperCollider use cases. The Bela setup is unique because it can run at very low latency (down to 8 sample buffers). Interestingly, at high latency, it can outperform systems with similar CPUs while using less power. The NanoPi has excellent performance for the price, and it can operate at low latency (a 64-sample buffer). The built-in outputs on the Raspberry Pis have somewhat limited buffer size options, only running at 512 and 1,024.

11.6 Case Studies with Small Computers and SuperCollider

In this section, you will find case studies demonstrating how you might use an ARM SBC (or SBCs) running SuperCollider. With reference to the features and capabilities discussed in previous sections, I will make recommendations as to the most appropriate ARM SBC for each case study.

11.6.1 Normal SuperCollider Development System

Given the performance of a Raspberry Pi 4 combined with an audio hat such as the HifiBerry, it would make an excellent SuperCollider development system. In this case, you would use the Pi just like any other computer: put it in a case, plug in a USB keyboard and mouse, and connect it to a monitor via HDMI.

11.6.2 The $150 Effects Pedal-Board

Guitarists often have a pedal-board, a board with several effects pedals sitting on it, all chained together. You can create a SuperCollider powered pedalboard using multiple ARM SBCs chained together. Figure 11.8 shows a mock-up of a pedalboard with four NanoPi Neo units.

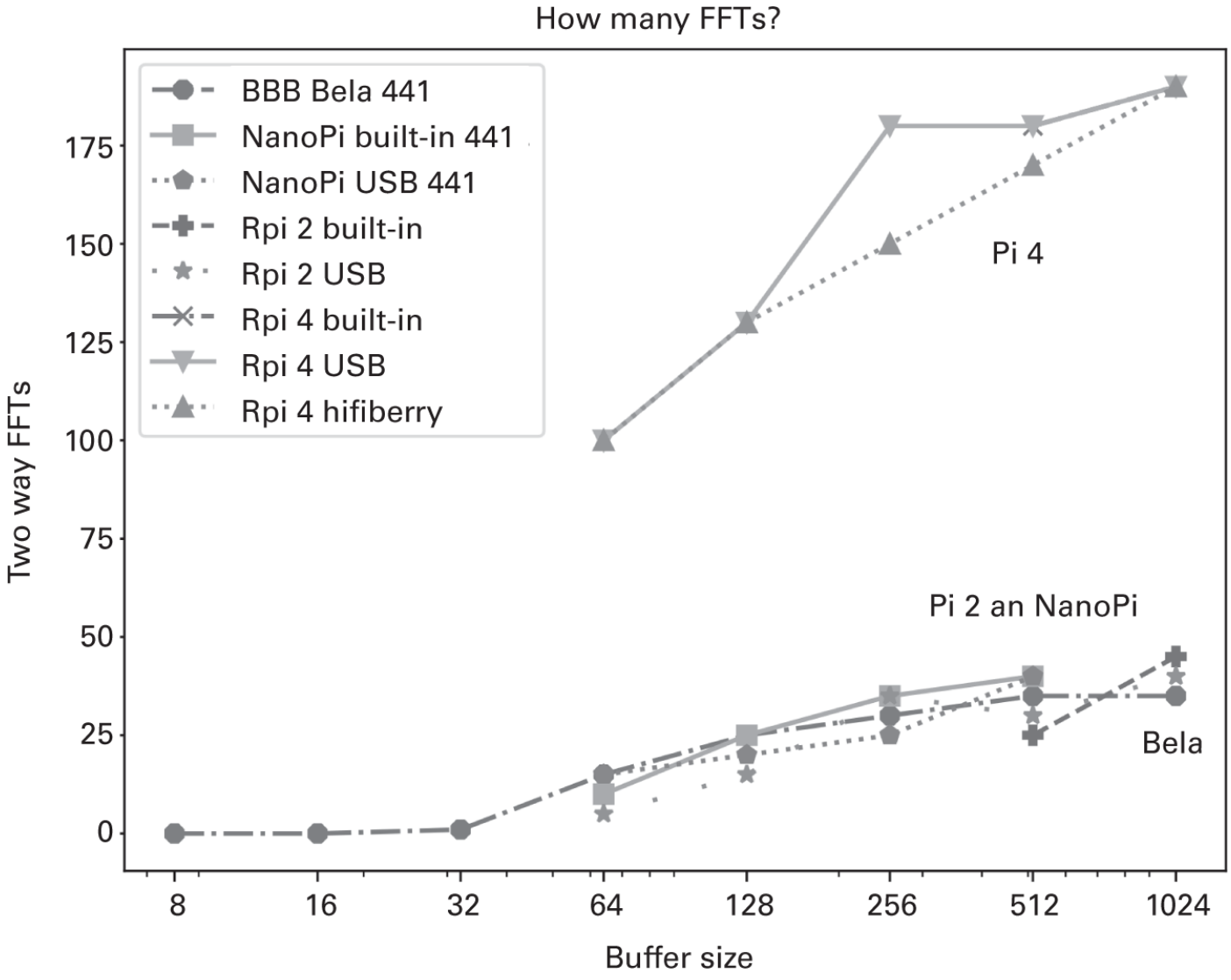


Figure 11.7
How many FFT synths each ARM SBC can run at a sample rate of 44,100 Hz and various buffer sizes.

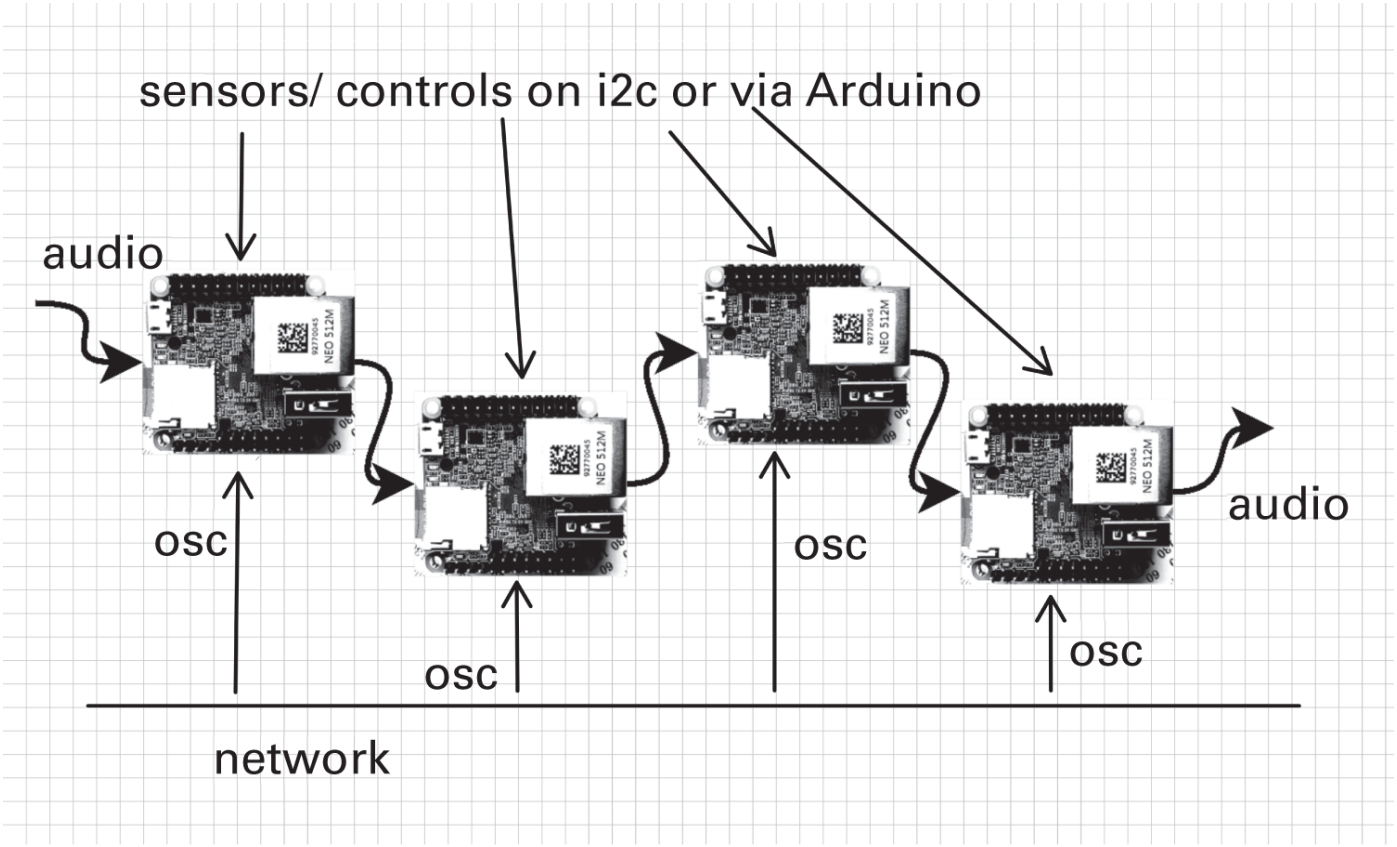


Figure 11.8
The $150 effects pedalboard, showing a mock-up of four NanoPi Neos that are connected.

The advantages of an ARM SBC SuperCollider effects pedal over a regular pedal are that it is cheaper and you can run any effect that you can program in SuperCollider from a simple flanger through to a complex FFT effects chain. The NanoPi is suitable for this application, as it is small, provides audio I/O without a daughterboard, and is quite performant at low latencies. You would need to interact with the scsynth servers running on the NanoPis to adjust the controls. There are a few ways that you might do that:

1. Send OSC messages from another machine running scide using the Server class.
2. Use a custom user interface that talks to scsynth or sclang using OSC. You could use TouchOSC[11] for this.
3. Using physical controls such as sensors wired directly into the GPIO pins on the NanoPi. The NanoPi does not have Arduino-like A/D converters for raw analog sensors, but you could use i2c compatible sensors.
4. Using an Arduino-type board with built-in sensors, such as the Seeed Studio Wio Terminal, communicating over the Serial bus with the NanoPi, or the Bakebit board (available as an add-on for the NanoPi).

Networking and power are things to consider with the pedalboard because it would be messy and inconvenient to have four network cables and four separate power bricks. You could use a USB power bank with multiple charge outputs to provide a neater power solution, and you could add USB Wi-Fi adapters to replace the network cabling. Even with all that taken into account, you could build all this for around $150.

11.6.3 Live Coding with Live Percussion

My primary personal use of ARM SBCs for SuperCollider has been the live coder and live percussionist setup shown in figure 11.9. Here, I have two musicians playing a drum kit and using live coding. The drum kit could be electronic, or it could be acoustic. A Beaglebone and Bela receive the audio from the drum kit and process it through SuperCollider. On the other side, a live coder performs in the usual way, but their code sends OSC messages to the ARM SBC to rhythmically adjust the processing applied to the drum kit. A USB connection from the live coding laptop to the Beaglebone provides power and networking.

The Beaglebone and Bela combination is appropriate here, as percussionists require very low latency to retain appropriate responsiveness from their instrument. The Bela audio system's high reliability and high fidelity mean it can be the only source of drum audio, leading to a simple audio setup.

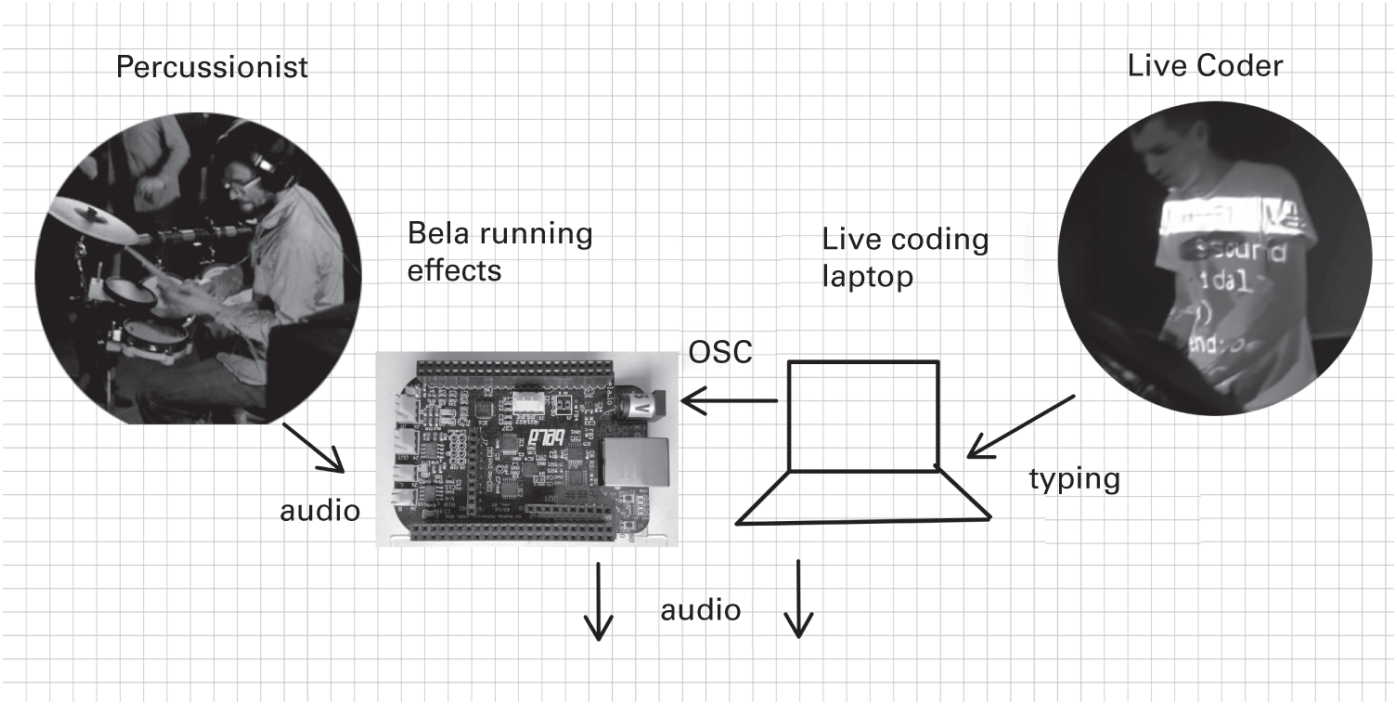


Figure 11.9
Live coding with live percussion setup. The live percussion is processed through a low-latency Bela setup, which is remotely controlled via OSC messages emitted from a live coding environment.

11.7 Conclusion

In this chapter, we have explored the use of SuperCollider on small computers. We have looked at four different ARM-based computer systems, each of which has different characteristics, as well as considering audio I/O daughterboards. We have seen that the systems are all capable of very usable DSP performance, but that there is some variation in the audio quality. We finished the chapter with some case studies showing how you might use an ARM SBC in your practice as a SuperCollider musician and we suggested which of the systems is most appropriate in each case. I have greatly enjoyed carrying out and presenting this investigation, and I hope that it inspires you to blow up a few boards and bend a few pins of your own.

Notes

1. Recent Mac Minis do have an ARM chip, but they do not meet the requirement of being cheap!
2. https://wiki.friendlyelec.com/wiki/index.php/NanoPi_NEO.
3. https://beagleboard.org/black/.
4. E.g., https://github.com/supercollider/supercollider/blob/develop/README_RASPBERRY_PI.md.
5. https://www.friendlyelec.com/.
6. https://monome.org/.
7. This is true unless you are using Linux on your main machine and you specify the -X option for SSH, but in my experience, that was very slow.
8. https://github.com/supercollider/supercollider/blob/17e0755310f3613891bc5850b208331e6ebfee65/README_RASPBERRY_PI.md.
9. https://github.com/supercollider/supercollider/blob/17e0755310f3613891bc5850b208331e6ebfee65/README_RASPBERRY_PI.md.
10. https://github.com/endolith/waveform_analysis.
11. https://hexler.net/touchosc.

References

Edstrom, B. 2016. *Arduino for Musicians: A Complete Guide to Arduino and Teensy Microcontrollers*. New York: Oxford University Press.

Marasco, A. T. 2022. "Approaching the Norns Shield as a Laptop Alternative for Democratizing Music Technology Ensembles." In *New Interfaces for Musical Expression*. https://doi.org/10.21428/92fbeb44.89003700.

McFee, B., C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. 2015. "librosa: Audio and Music Signal Analysis in Python." In *Proceedings of the 14th Python in Science Conference*, pp. 18–25.

Richards, J. 2017. "DIY and Maker Communities in Electronic Music." In *The Cambridge Companion to Electronic Music* (2nd ed.), ed. N. Collins and J. d'Escriván (pp. 238–257). Cambridge: Cambridge University Press.

Singh, K. J., and D.S. Kapoor. 2017. "Create Your Own Internet of Things: A Survey of IoT Platforms." *IEEE Consumer Electronics Magazine*, 6(2): 57–68.

Süzen, A. A., B. Duman, and B. Şen. 2020. "Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI Using Deep-CNN." In *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*.

Wang, G., G. Essl, and H. Penttinen. 2014. "The Mobile Phone Orchestra." In *The Oxford Handbook of Mobile Music Studies,* Vol. 2, ed. S. Gopinath and J. Stanyek. Online edition, Oxford Academic. Accessed April 28, 2023, from https://doi.org/10.1093/oxfordhb/9780199913657.013.018.

12 The SuperCollider GUI Library

Eli Fieldsteel

12.1 GUI Overview

The term *graphical user interface* (GUI) refers to an arrangement of interactive visual objects, collectively serving as an intermediary layer between user and code and providing an intuitive system for controlling a program. In addition to various sound-specific classes, SuperCollider contains a diverse family of GUI classes, such as knobs, sliders, and buttons. A GUI is not always necessary, but it can be helpful in certain cases, like emulating a hardware synthesizer, creating learning tools for students, or sending code to collaborators who are unfamiliar with text-based programming. A well-designed GUI presents a program's functionality in clear, accessible terms, highlighting the essential modes of interaction while concealing the underlying framework.

This chapter introduces the fundamentals of working with GUIs, beginning with essential attributes and behaviors and progressing through some higher-level concepts, such as external interaction, organizational strategies, and custom graphics.

12.2 GUI Basics

12.2.1 Windows

A GUI begins with a new `Window`, which provides a rectangular space on which GUI objects can be placed. When creating a window, we provide a title for the window's top bar, and the window's bounds (i.e., its size and location) specified as an instance of `Rect`. A Rect contains four values. The first two numbers represent the cartesian point (horizontal, vertical) that coincides with the window's bottom-left corner, measured in pixels relative to the bottom-left corner of the screen. The second two numbers determine the window's width and height. Additional Boolean values (`true` or `false`) can be supplied to determine whether the window is resizable and whether it has a border (i.e., a menu bar). Making these arguments `false` is useful for fixing the window on a particular part of the screen, which prevents a user from moving it. Newly created windows are invisible and can be made visible using the message `front`.

A window will remain visible above other windows if its `alwaysOnTop` attribute is `true`. This is sometimes convenient during development to avoid having to toggle back and forth between the window and the code environment. Windows can be hidden/unhidden by setting `visible` to `false`/`true`. A window can be removed with `close`, equivalent to clicking its close button. Borderless windows can be closed only via this message. Closing a window is permanent; once closed, a window cannot be accessed. Whether a window is closed can be queried with `isClosed`, which can be useful in conditional statements to determine whether window interaction should be attempted (see figure 12.1).

```
(
w = Window("empty window", Rect(50, 100, 400, 300));
w.front;
)
w.alwaysOnTop_(true);
w.visible_(false);
w.isClosed; // returns false; invisible ≠ closed
w.visible_(true);
w.close;
w.visible_(true); // no effect after closing
w.isClosed; // returns true
```

Figure 12.1
Basic creation and manipulation of a window.

The class method `closeAll` is a useful cleanup utility if you find yourself inundated with redundant window copies. Invoking this method at the top of your GUI code can help avoid accidental window buildup. The class method `screenBounds` returns a `Rect` that represents the size of your screen. Strategic use of this method, combined with `Rect`'s `width` and `height` methods, can help ensure uniform positioning across different screen sizes.

```
(
Window.closeAll; // close any previously opened windows
// make a centered window
w = Window(
   "centered window",
   Rect(
        Window.screenBounds.width/2-200,
        Window.screenBounds.height/2-150,
        400,
        300
```

```
    )
).front;
)
```

Figure 12.2
Practical usage of `closeAll` and `screenBounds`.

12.2.2 Views

`View` is the main parent class of GUI objects, and it is also the general term used to describe these objects. Upon creation, a view requires two things: its parent (i.e., the space on which the view will reside) and its bounds, which determines the rectangular space that the view will occupy. Unlike windows, whose bounds are measured from the *bottom*-left screen corner, the bounds of a view are measured from the *top*-left corner of its parent. In the SC GUI system, the rectangular space on the body of a window is itself, in fact, a type of view (called a `TopView`). SC's various container views can also be valid parents (See section 12.2.8.) A list of GUI objects can be found in the Help file, "List of GUI classes."[1] Table 12.1 lists a subset of commonly used views.

Table 12.1
Names and descriptions of commonly used view subclasses

Class Name	Description
`Button`	Multistate button
`Knob`	Rotary controller
`ListView`	Display list of selectable items
`MultiSliderView`	Bank of multiple sliders
`NumberBox`	Modifiable field for numerical values
`PopUpMenu`	Drop-down menu for individual item selection
`RangeSlider`	Slider with an extendable handle on each end
`Slider`	Linear controller
`Slider2D`	Two-dimensional (2D) slider
`StaticText`	Noneditable text display; useful for labels
`TextField`	Simple editable text display
`TextView`	Editable, formattable, multiline text display

```
(
w = Window(
        "window with a view",
        Rect(50, 100, 400, 300)
).front;
Slider(w, Rect(50, 50, 300, 50));
)
```

Figure 12.3
Creation and placement of a `Slider` view in a parent window.

12.2.3 Layout Management

Layout is among the most potentially tedious aspects of building a GUI. Without helpful tools for layout management, positioning views on a window can devolve into tiresome pixel hunting. To facilitate this process, the SC GUI system features a small family of auto-layout tools, which are broadly discussed in the "Layout Management" guide file.[2] `HLayout` and `VLayout` support horizontal/vertical arrangements, `GridLayout` enables grid-based arrangements, and `StackLayout` allows multiple views to be superimposed, with the ability to select dynamically which view is displayed above the others. A primary benefit of these tools is that they attempt to make smart decisions about space allocation based on the type and content of each view, helping us avoid getting bogged down in details. Additionally, these tools automatically enlarge a parent window if it is too small to accommodate its children and will dynamically adjust views when the parent window is resized. Furthermore, `HLayout` and `VLayout` can be nested inside one another to create custom grid-style arrangements. Basic usage of these techniques is demonstrated in figure 12.4. Note that we do not provide bounds information for the individual views, as the `Layout` object will automatically do so. Additional options for fine-tuning the smart behavior of these auto-layout classes are documented in their respective Help files.

```
(
Window(
   "horizontal/vertical layout",
   Rect(100, 100, 250, 300)
).layout_(VLayout(
   Button(),
   HLayout(Slider(), Slider(), Slider(), Slider())
)
).front;
Window(
   "grid layout",
   Rect(400, 100, 200, 100)
).layout_(GridLayout.rows(
    [Button(), Button(), Button(), Button()],
    [Knob(), Knob(), Knob(), Knob()],
)).front;
)
// try resizing windows with the mouse to see dynamic adjustments
```

Figure 12.4
Usage of auto-layout tools

12.2.4 Getting and Setting Attributes

Information about the current state of a view is stored in a collection of variables, or attributes, associated with the view instance. The `View` class defines attributes common to all its subclasses, such as background color, bounds, and visibility. Individual subclasses have additional sets of attributes that are self-specific. As an example, `Slider` and `RangeSlider` have an attribute called `orientation`, which can be set to `\horizontal` or `\vertical`, but this attribute does not apply to most other types of views. When dealing with GUI objects and their behaviors, we frequently need to retrieve and/or alter these attributes. Many of these follow SC's standard "getter/setter" syntax, which appears in figure 12.5, and specific cases of getting/setting window attributes appear in figure 12.6.

```
myView.parameter; // retrieve, or "get" parameter value
myView.parameter_(newValue); // update, or "set" parameter   value
 to "newValue"
```

Figure 12.5
Generalized syntax for getting and setting.

```
(
w = Window(
   "window with a view",
   Rect(50, 100, 400, 300)
).front;
~slider = Slider(w, Rect(50, 50, 300, 50))
.background_(Color.rand);
)
~slider.background; // get background color
~slider.background_(Color.green); // set background color to green
~slider.bounds_(Rect(50, 200, 300, 50)); // reposition slider
~slider.visible_(false); // make invisible
```

Figure 12.6
Getting and setting attributes of a slider.

As a side note, color, introduced in figure 12.6, is expressed using the `Color` class. A default instance of `Color` contains four values: red, green, blue, and an "alpha" transparency value, each between 0 and 1. Alpha defaults to 1 (opaque) if unspecified. `Color.new(0, 0.5, 1)`, for instance, corresponds to a cornflower blue. Color can

also be specified using integers ranging from 0 to 255 (e.g., `Color.new255(0, 127, 255)`) or a hexadecimal string, e.g. `Color.fromHexString("#007FFF")`. Other useful methods are detailed in the Color Help document.[3]

12.2.5 Values and Actions

We often want a view to take some action in response to user input, and we usually want that action to depend on the new state of the view. As a simple example, consider using the slider in figure 12.3 to adjust the amplitude of a signal so that it is silent when the slider is on the left and nominal on the right, with continuous changes in between. The state of a slider (and most other views) is stored in an attribute called `value`. A view also includes an `action` attribute, which defines a function to be evaluated in response to user input. An argument declared in the action function represents the view itself, thus enabling direct access to its attributes. It is possible to simulate user interaction using the `valueAction` setter method, which updates a view's state and also immediately calls its action. Using the `value` method as a setter, on the other hand, updates a view's state but does not execute its action. Figures 12.7 and 12.8 demonstrate simple approaches for controlling sound with a slider and a button, respectively. Combining these two views into a single window is left as a suggested exercise for the reader.

```
s.boot;
(
~amp = 0.5;
~synth = {|amp| PinkNoise.ar(0.3!2) * amp} .play(args:   [\amp, ~am
p]);
~slider = Slider()
.orientation_(\horizontal)
.value_(~amp)
.action_({|v|
   ~amp = v.value;
   ~synth.set(\amp, ~amp);
});
w = Window(
   "slider sound control",
   Rect(50, 100, 400, 50)
)
.layout_(HLayout(~slider))
.onClose_({~synth.set(\gate, 0)})
.front;
)
~slider.valueAction_(rrand(0.0, 1.0)); // manually update view &
```

```
 invoke action
w.close; // also releases the synth because of onClose above
```

Figure 12.7
Controlling signal amplitude with a slider.

Newer users should take note of the syntax for establishing a button's states, as shown in figure 12.8. A button can have one or more states. A simple on/off toggle button, for instance, has two states, referred to by their indices, 0 and 1. A button's `states` are defined by providing an array containing internal arrays, one for each state. Each internal array contains text to display, a text color, and a background color for the state that it represents. Also note the use of the `Font` class, which is relevant for any views that include text. A new instance of `Font` expects a string for the font name, followed by the font size.

```
s.boot;
(
~amp = 0;
~synth = {|amp| PinkNoise.ar(0.3!2) * amp} .play(args: [\amp,   ~am
p]);
~button = Button()
.font_(Font("Courier", 24))
.states_([// an Array with [text, text color, background color]   f
or each state
   ["OFF", Color.gray(0.3), Color.gray(0.75)], // state 0
   ["ON", Color.white, Color(0, 0.5, 1)] // state 1
])
.action_({|v|
   ~amp = v.value;
   ~synth.set(\amp, ~amp)
}); // button state used as amplitude value
w = Window(
   "button sound control",
   Rect(50, 50, 250, 100)
)
.layout_(HLayout(~button))
.onClose_({~synth.set(\gate, 0)})
.front;
)
~button.valueAction_(1-~button.value); // manually toggle button
 state & invoke action
w.close; // also releases the synth because of onClose above
```

Figure 12.8
Controlling signal amplitude with a button.

12.2.6 Range Mapping

The default value range of sliders, knobs, and other numerical views is 0 to 1. This may work well enough for signal amplitude, but for frequency and other synthesis parameters, these values must be mapped to a different range before they can be useful. Several range-mapping methods exist to handle such conversions. Generally, these methods accept four essential arguments: the min/max of the input range, and the min/max of the output range. Some of these methods are provided in table 12.2, and the SimpleNumber Help file includes a more complete listing.[4] Note that if the input or output range is exponential (`exp`), then both `min` and `max` must have the same sign, and neither can be 0. Figure 12.9 manipulates slider values with `lincurve` to produce a more exponentially shaped amplitude curve, and thus (arguably) a more natural-sounding response.

Table 12.2
A partial list of methods for mapping one value range to another

Method	Description
`linlin()`	Linear to linear range.
`expexp()`	Exponential to exponential range. Neither range may include 0.
`linexp()`	Linear to exponential range. Output range may not include 0.
`explin()`	Exponential to linear range. Input range may not include 0.
`lincurve()`	Linear to flexibly curved range. Curve parameter determines the shape of the output range: 0 is linear, and positive or negative values bend the curve up or down, respectively.
`curvelin()`	Flexibly curved to linear range. Curve parameter determines the shape of the output range: 0 is linear, and positive or negative values bend the curve up or down, respectively.

```
s.boot;
(
~amp = 0.5.lincurve(0, 1, 0, 1, 3); // 3 represents the   curvature
of the mapping
~synth = {|amp| PinkNoise.ar(0.3!2) * amp} .play(args: [\amp,   ~am
p]);
~slider = Slider()
.orientation_(\horizontal)
.value_(0.5)
.action_({|v|
```

```
    ~amp = v.value.lincurve(0, 1, 0, 1, 3);
    ~synth.set(\amp, ~amp)
});
w = Window(
    "lincurve amplitude mapping",
    Rect(50, 100, 400, 50)
)
.layout_(HLayout(~slider))
.onClose_({~synth.set(\gate, 0)})
.front;
)
w.close;
```

Figure 12.9
Using `lincurve` to map amplitude values nonlinearly.

Alternatively, value mapping can be handled by `ControlSpec`, a utility class designed to convert numbers back and forth between 0–1 and another custom range. Primarily, `a ControlSpec` requires min/max values and a warp parameter, which can be a symbol like `\lin` or `\exp`, or a number (as demonstrated with `lincurve` in figure 12.9). Once created, `map` and `unmap` translate between ranges. Commonly used ranges are available as predefined specs, the full list of which can be obtained by evaluating `ControlSpec.specs.keys`. We can use one of these presets by calling `asSpec` on the appropriate symbol (e.g., `\freq.asSpec`). Figure 12.10 replaces the previous amplitude slider with a frequency slider and uses `ControlSpec` to manage the range mapping.

```
s.boot;
(
~freqspec = ControlSpec(60, 500, \exp);
~freq = ~freqspec.map(0.5);
~synth = {|freq| Saw.ar(freq.lag(0.1) + [0, 1], 0.03)}    .play(arg
s: [\freq, ~freq]);
~freqslider = Slider()
.orientation_(\horizontal)
.value_(0.5)
.action_({|v|
    ~freq = ~freqspec.map(v.value);
    ~synth.set(\freq, ~freq);
});
w = Window(
    "ControlSpec frequency mapping",
```

```
    Rect(50, 100, 400, 100)
)
.layout_(HLayout(~freqslider))
.onClose_({~synth.set(\gate, 0)})
.front;
)
w.close;
```

Figure 12.10
Using `ControlSpec` to map frequency values.

Note that when using a GUI to control sound, applying a lag time to synthesis arguments can help produce a cleaner sound. When lagged, parameters smoothly interpolate from one value to the next, even if an instantaneous change occurs in a view's state. In the absence of interpolation, sudden changes to a synthesis parameter may result in a hard click or zipper noise.

12.2.7 EZGui

Graphically modeling a single piece of data is often enhanced by using a small cluster of views. For example, we might want to control a frequency parameter using a slider but also use a number box to display the value and a text object to provide a label. For this purpose, there is a family of EZ GUI objects (listed in table 12.3), which bundle multiple objects together and present them as one. This term is a stylized spelling of the word "easy," meant to indicate that these classes simplify the GUI building process. As an added bonus, EZ GUI objects generate their own parent window, if none is provided (compare `Slider.new` to `EZSlider.new`). Figure 12.11 modifies the code in figure 12.10 by substituting an EZSlider. Note that in this example, we let the EZSlider create its own window, but we can still access this window by calling `window` on the EZ object.

Table 12.3
Names and descriptions of commonly used EZ-GUI classes

Class Name	Description
`EZKnob`	Knob, NumberBox, and optional StaticText
`EZListView`	ListView and optional StaticText
`EZNumber`	NumberBox and optional StaticText
`EZPopUpMenu`	PopUpMenu and optional StaticText
`EZRanger`	RangeSlider, NumberBox, and optional StaticText Slid
`EZSlider`	Slider, NumberBox, and optional StaticText

```
s.boot;
(
~freqspec = ControlSpec(60, 500, \exp);
~freq = ~freqspec.map(0.5);
~synth = {|freq|
   Saw.ar(freq.lag(0.1) + [0, 1], 0.03)
}.play(args: [\freq, ~freq]);
~freqslider = EZSlider(
   label: "freq: ",
   controlSpec: ~freqspec,
   initVal: 0.5
)
.setColors(Color.gray(0.4), Color.gray(0.75))
.value_(~freq)
.action_({|v|
   ~freq = v.value;
   ~synth.set(\freq, ~freq);
})
.onClose_({~synth.set(\gate, 0)});
)
~freqslider.window.close;
```

Figure 12.11
Controlling a frequency parameter using EZSlider.

12.2.8 Hierarchical Views

The figures presented so far are relatively simple, each featuring only a handful of views. In practice, however, a developed GUI may have dozens or even hundreds of views. To efficiently manage such a large collection, a higher-level organizational structure becomes helpful. If every view uses the main window as its parent, there is no sense of hierarchy. While it is possible to create an organizational structure of your own (e.g., storing each view in a multidimensional array), the parent/child aspect of view creation implicitly provides an organizational structure of its own. Just as every view has a parent (with the exception of the topmost view, whose parent is `nil`), every view can also have some number of child views placed upon it. In fact, a view is sometimes called a "container," a nod to the fact that it is a storage device for other views. An array of a view's children is gettable with the `children` method. The order of this array is determined by the order in which the child views were created.

To put this approach into practice, we typically create new instances of the `View` class, using the main window as their parent, and each serves to delineate a subsection of the larger interface (e.g., oscillator section, filter section). Additional views and

subviews can be placed upon these sections, creating as many hierarchical layers as desired. One benefit of this approach is that a message to a view will automatically propagate to its children, which relay the message to their children, and so on. Thus, a singular expression can be used to set the attributes for a large collection of objects. Relying on this structure also means that GUI objects no longer require quite as fine-grained a naming scheme. Despite the fact that the individual sliders, buttons, and knobs are unnamed, we can still access them via their parents. In figure 12.12, for example, the rightmost slider can be set to a random value with `~sliders.children[3].valueAction_(rrand(0.0, 1.0))`. Similarly, we can hide the knobs by changing the visibility of their parent (e.g., `~knobs.visible_(false)`).

```
(
~sliders = View()
.background_(Color.rand)
.layout_(HLayout(
   Slider(), Slider(), Slider(), Slider()
));
~buttons = View()
.background_(Color.rand)
.layout_(HLayout(
   Button(), Button(), Button(), Button()
));
~knobs = View()
.background_(Color.rand)
.layout_(HLayout(
   Knob(), Knob(), Knob(), Knob()
));
~mainView = View()
.background_(Color.gray(0.5))
.layout_(VLayout(~buttons, ~sliders, ~knobs));
w = Window.new("hierarchical views", Rect(100,100,300,400))
.layout_(VLayout(~mainView))
.front;
)
```

Figure 12.12
Using subviews to create and organize hierarchical sections of a GUI.

12.2.9 External GUI Interaction

Options for interacting with GUI extend beyond clicking and dragging with the mouse. A view can perform an action if the mouse is simply hovering over it, and actions can be registered in response to computer keyboard input. GUI objects also can be manipulated

by MIDI or OSC controllers, and state changes can be scheduled using a `Routine`. These interactive augmentations can add feature richness that not only looks nice, but also helps clarify the functionality and draws the user's attention toward specific features.

All views respond to a set of methods that define a function to be evaluated when certain mouse/keyboard events occur, such as mousing over a view or pressing a specific key. These methods are detailed in the "Key and Mouse Event Processing" section of the View Help document.[5] Compared to a view's normal action function, more arguments are passed to these functions, which provide access to all the relevant information about the mouse/keyboard event. Mouse action functions take three arguments: the view instance and the pixel coordinates of the mouse. Keyboard functions take six arguments: the view, the character, information on whether modifier keys are pressed, a unicode integer, a hardware-dependant keycode, and the key identifier defined by the Qt framework (the underlying library used to implement SC GUIs). In practice, it is rare to need all (or even most of) these arguments. The character alone is often sufficient to define the desired action. In SC, characters are implemented with the `Char` class, denoted with a preceding dollar sign (e.g., `$Q`, `$5`, `$&`). Figure 12.13 illustrates basic usage of custom keyboard/mouse actions.

```
(
var active = true;
w = Window("mouse/keyboard interaction", Rect(50, 100, 400, 200))
.acceptsMouseOver_(true) // necessary in order to use   mouseOverAc
tions
.layout_(VLayout(
   Slider(), Slider(), Slider(),
   StaticText()
   .string_("spacebar to disable/enable")
   .align_(\center)
))
.front;
// set the actions for the sliders (children 0-2)
w.view.children[0..2].do({|sl, i|
   sl.action_({|v| [i, v.value].postln})
   .orientation_(\horizontal)
   .mouseOverAction_({|v| v.background_(Color(0.7, 0.8, 0.9))})
   .mouseLeaveAction_({|v| v.background_(Color.gray(0.75))});
});
w.view.keyDownAction_({|v, char|
   if(char == $) // space character
   {
```

```
    active = not(active);
    w.view.children[0..2].do({|sl|
    sl.enabled_(active);
    sl.background_(Color.gray (if(active, {0.75}, {0.65})))
    });
    }
});
)
```

Figure 12.13
Defining custom actions to be invoked in response to mouse/keyboard input.

Some key-responsive actions are prebuilt into the GUI framework. Pressing the *Tab* key, for example, will cycle the focus through available views, indicated by a thin outline. When in focus, most views perform their primary action if the spacebar is pressed. When a slider or knob is in focus, pressing the *c* key will center the handle, pressing *n* or *x* will jump the slider to its mi*n*imum or ma*x*imum position, and *r* will randomize the value, all invoking the view's action.

An external controller, such as a MIDI keyboard or mobile device transmitting OSC messages, can manipulate a GUI and invoke its actions, perhaps for display purposes. Generally, this workflow involves a responder object (e.g., `MIDIdef` or `OSCdef`), which receives external data and passes them to a view by calling `valueAction`. (See chapter 4 for a more detailed discussion of MIDI and OSC functionality.) GUI updates are considered low priority compared to synthesis and event timing, and they cannot be manipulated directly from certain contexts, such as from within a MIDI/OSC responder. Attempts to do so produce an error that reads, "You can not use this Qt functionality in the current thread. Try scheduling on AppClock instead." A common solution is to enclose the problematic GUI code in a function and use the `defer` method, as shown in figures 12.14 and 12.15. This solution is also applicable when scheduling GUI actions with `TempoClock` or `SystemClock`.

```
(
var slider, spec;
spec = ControlSpec.new(0, 16383, \lin);
MIDIIn.connectAll;
MIDIdef.bend(\pitchbend, {|bendval|
    {slider.valueAction_(spec.unmap(bendval))} .defer;
});
slider = Slider()
    .acceptsMouse_(false) // user can't change; for display only
    .value_(0.5)
    .action_({|v| v.value.postln;});
```

```
w = Window.new("MIDI pitch bend", Rect(100,100,200,400))
.onClose_({MIDIdef(\pitchbend).free})
.layout_(HLayout(slider))
.front;
)
MIDIIn.doBendAction(1, 0, 100); // spoof a pitch bend message
```

Figure 12.14
Controlling a view with MIDI pitch bend data.

```
(
var button, knob, sched, i = 0;
knob = Knob()
.canFocus_(false).acceptsMouse_(false); // user can't control   it
 directly
button = Button()
.states_([
   ["automate", Color.black, Color.gray(0.75)],
   ["automate", Color.white, Color(0, 0.5, 1)]
])
.action_({|v|
   if(v.value == 1) {
   sched = Routine({
        loop{
              i = i + 0.02;
              {knob.valueAction_(i.fold(0, 1))} .defer;
              wait(1/30);
        }
   }).play;
   }
   {sched.stop};
});
w = Window.new("GUI automation", Rect(100, 100, 200, 300))
.onClose_({sched.stop})
.layout_(VLayout(button, knob))
.front;
)
```

Figure 12.15
Scheduling GUI updates with a `Routine`.

12.3 Custom Graphics with UserView and Pen

Adding custom graphics to a GUI is rarely necessary, but it can be useful for adding emphasis, re-creating the physical appearance of other hardware/software, or just making an interface a little more exciting. This ability is provided by `UserView`, an otherwise featureless view on which lines, curves, and shapes can be drawn. Instructions for what to draw on a UserView are contained in a function, stored in its `drawFunc` attribute.

Within a drawFunc, `Pen` is the primary tool for creating custom graphics. `Pen` is a slightly unusual class, in that we do not create new instances of it, as we do with `Window.new`, `Slider.new`, etc. Conceptually, there is only one `Pen`—the class itself—which we control using class methods that influence its attributes and movements. Table 12.4 lists some basic `Pen` methods, and the `Pen` Help file contains a more comprehensive listing, along with many colorful examples.[6]

Table 12.4
Commonly used class methods for `Pen`

Method	Description
`Pen.width_(n)`	Set line/curve thickness to *n* pixels.
`Pen.strokeColor_(aColor)`	Set the color of drawn lines/curves.
`Pen.fillColor_(aColor)`	Set the color used to fill closed paths.
`Pen.moveTo(x@y)`	Move the pen to the point (*x, y*).
`Pen.lineTo(x@y)`	Construct a line from the current coordinate to the point (x, y). After this, the pen coordinate is (x, y).
`Pen.line(x@y, z@w)`	Construct a line between the two specified points. After this, the pen coordinate is (z, w).
`Pen.addArc(x@y, r, p, q)`	Construct a circular arc around the point (x, y) with radius r (pixels), starting at angle p and rotating by q (radians). $q = 2\pi$ draws a complete circle.
`Pen.addWedge(x@y, r, p, q)`	Construct an arc wedge (i.e., a section of a pie) around the point (x, y) with radius r (pixels), starting at angle p and rotating by q (radians).
`Pen.addRect(aRect)`	Construct the specified rectangle.
`Pen.stroke`	Render all lines, curves, arcs, etc. previously constructed. After stroking, the pen position is reset to (0, 0).
`Pen.fill`	Render the insides of closed paths. An unclosed path will be filled as if its end points were connected by a line.
`Pen.fillStroke`	Combination of `Pen.stroke` and `Pen.fill`.

12.3.1 Lines, Curves, and Shapes

When using `Pen`, there is a distinction between "constructing" and "drawing" a path. The workflow involves constructing paths first, through specification of lines and other shapes. These constructions are not visually rendered until the `stroke` and/or `fill` methods are called. Thus, the order in which instructions are implemented in a

drawFunc is significant. For example, to create a pair of lines in two different colors, we must set the color, construct the first line, and stroke it. Then, we change the color and construct/stroke the second line.

```
(
w = Window("basic Pen usage", Rect(100, 100, 300, 300)).front;
u = UserView(w, w.view.bounds)
.background_(Color.white)
.drawFunc_({
   Pen.width_(5);
   // Line
   Pen.strokeColor_(Color.black);
   Pen.moveTo(105@20);
   Pen.lineTo(30@80);
   Pen.stroke; // <—draw line
   // Rectangle
   Pen.fillColor_(Color.red(0.8, 0.5));
   Pen.width_(2);
   Pen.addRect(Rect(160, 55, 100, 80));
   Pen.fill; // <—draw rectangle
   // Wedge
   Pen.strokeColor_(Color(0, 0.5, 1));
   Pen.fillColor_(Color.gray(0.8));
   Pen.addWedge(100@200, 70, 0, 3pi/2);
   Pen.fillStroke; // <—draw wedge
});
)
```

Figure 12.16
Basic usage of Pen and UserView.

Creating a uniquely designed button is a simple but reasonably practical application. If we restrict ourselves to the Button class, a rectangular shape is our only option. Figure 12.17 presents an alternative approach, which iterates over a collection of points to create a hexagon. Using techniques from figure 12.13, we use mouseDownAction to provide the UserView with buttonlike behavior.

```
(
var on = false;
w = Window("custom button with Pen", Rect(50, 100, 300, 200)).   fr
ont;
u = UserView(w, Rect(100, 50, 100, 100))
```

```
.background_(Color.clear)
.drawFunc_({
   Pen.strokeColor_(Color.black);
   Pen.fillColor_(
        Color(0, 0.75, 1, if(on, {0.8}, {0.2}))
   );
   Pen.width_(3);
   Pen.moveTo(50@4);
   [10@28, 10@72, 50@96, 90@72, 90@28, 50@4].do({|n| Pen.   lineTo
(n)});
   Pen.fillStroke;
   Pen.stringCenteredIn(
        if(on, {"ON"}, {"OFF"}),
        Rect(0, 0, 100, 100),
        Font("Arial", 24),
        Color.black
   );
})
.mouseDownAction_({|view|
   on = not(on);
   if(on, {"on action".postln}, {"off action".postln});
   view.refresh;
})
)
```

Figure 12.17
A custom button with `UserView` and `Pen`.

12.3.2 Animation

The `refresh` method causes a `UserView` to reevaluate its `drawFunc` and redraw its image. By calling `animate_(true)`, a `UserView` will repeatedly refresh itself at its `frameRate`. In figures 12.16 and 12.17, animating would do nothing of interest because `drawFunc` produces the same result each time. But, if `drawFunc` produces variable results, we can create animations. A common approach involves maintaining a global counter, incremented on each frame. This counter is then integrated with values that determine `Pen` behavior. Figure 12.18 uses a counter, bounded between 0 and 2π, to draw three concentric circles whose radii and colors fluctuate. By default, a `UserView` will clear itself before each new frame is drawn, but this is configurable by setting `clearOnRefresh` to `true` or `false`. If we make this attribute `false` and draw a partially transparent background layer at the start of each frame, the animation produces a visual delay effect.

```
(
var t = 0, rad;
w = Window("ripple", Rect(100, 100, 400, 400))
.front;
u = UserView(w, w.view.bounds)
.background_(Color.black)
.drawFunc_({
   Pen.fillColor_(Color.gray(0.0, 0.05));
   Pen.fillRect(u.bounds);
   Pen.width_(3);
   t = t + (2pi/200) % 2pi;
   rad = cos(t + [0, pi/10, pi/5]) / 2 + 0.5;
   rad.do({|n|
        Pen.strokeColor_(Color(0, n, rrand(0.7, 0.9)));
        Pen.addArc(200@200, n * 150 + 10, 0, 2pi);
        Pen.stroke;
   });
})
.clearOnRefresh_(false)
.animate_(true);
)
```

Figure 12.18
Creating an animated effect with `UserView` and `Pen`.

It is important to keep in mind that SuperCollider is not nearly as optimized for visuals as it is for sound. While it can often handle thousands of UGens with relative ease, overambitious use of animation can bring the frame rate to a crawl and overwhelm your computer's processing capabilities. These animation techniques are best used in moderation.

12.4 Managing GUI Updates and Communication

12.4.1 Observers and Dependancies

For users who are new to building GUIs, there may be a temptation to treat a view, such as a slider or knob, as being indistinguishable from or interchangeable with the data that it represents, e.g., calling `value` on the view whenever data is needed, rather than explicitly storing the data in a separate container. This approach is likely to become messy, especially when working on a large and complex interface, or building a system in which several views work together to represent a multifaceted piece of data. At worst, two views may inadvertently call `valueAction` on each other, resulting in an

infinite loop and a program crash. It is crucial to remember that a view itself is *not* the same thing as the data is models; rather, it is a separate mechanism for manipulating and/or displaying that data.

A much better practice involves registering views as "dependants" or "observers" of a piece of data. When the data change, they broadcast the changes to observers, who respond by refreshing themselves to reflect any changes. Thus, the source of a change can come from anywhere (including the same view that displays the data!), and the change will automatically propagate throughout the system. There are numerous variations on this observer "design pattern" throughout the programming world; one of the most common is called MVC, or Model-View-Controller), but they all share the same general principle of broadcasting data changes to registered observers.

As an example, consider a GUI that models a basic amplitude envelope (figure 12.19). In addition to using a graphical view of points connected by lines, we might want the option of individual sliders for controlling the attack time, release time, and peak level. These two interfaces are separate, but they represent the same information, so the observer pattern is a sensible choice here. In SC, the workflow involves first creating the data itself (`Env` is the most obvious choice). We then create a GUI updater function and register it as a `dependant` of the envelope. Any action that modifies the envelope is followed by a `changed` message, which alerts `dependants` that a change has occurred. When a change occurs, the data pass into the updater function as an argument, so that the data becomes accessible to the `dependants`.

```
s.boot;
(
~env = Env([0, 1, 0], [0.5, 0.5], \lin);
~timeSpec = ControlSpec(0.001, 1, \lin); // avoid zero-duration   e
nv segments
~envView = EnvelopeView().setEnv(~env)
.keepHorizontalOrder_(true) // prevent points from crossing one   a
nother
.setEditable(0, false).setEditable(2, false) // only the middle   p
oint is editable
.action_({|v|
   ~env.levels_([0, v.value[1][1], 0]);
   ~env.times_([v.value[0][1], 1-v.value[0][1]]);
   ~env.changed; // broadcast change to dependants
});
~envSliders = [
   Slider().value_(0.5),
   Slider().value_(1),
   Slider().value_(0.5)
```

```
].do({|v|
   v.action_({
   defer {// update data
        ~env.levels_([0, ~envSliders[1].value, 0]);
        ~env.times_([
              ~timeSpec.map(~envSliders[0].value),
              ~timeSpec.map(~envSliders[2].value)
        ].normalizeSum);
   };
   ~env.changed; // broadcast change to dependants
   });
});
w = Window.new("envelope GUI", Rect(100, 100, 300, 400))
.layout_(VLayout(
   ~envView,
   HLayout(~envSliders[0], ~envSliders[1], ~envSliders[2]),
   HLayout(
   StaticText().string_("attack").align_(\center),
   StaticText().string_("peak level").align_(\center),
   StaticText().string_("release").align_(\center)
   )
))
.onClose_({~env.removeDependant(~update); r.stop;}) // cleanup
.front;
~update = {|env|
   defer{
   ~envView.setEnv(env);
   ~envSliders[0].value_(~timeSpec.unmap(env.times[0]));
   ~envSliders[1].value_(env.levels[1]);
   ~envSliders[2].value_(~timeSpec.unmap(env.times[1]));
   };
};
~env.addDependant(~update);
r = Routine({
   loop{
   {SinOsc.ar([300, 310]) * ~env.kr(2) * 0.2} .play(fadeTime:0);
   1.wait;
   };
}).play;
)
```

Figure 12.19
An envelope GUI that uses a `dependancy` structure.

The curious reader may also wish to explore `NotificationCenter`, a utility class that facilitates `dependancy`-based design patterns with a more unified approach.

12.4.2 Constraining GUI Refresh Rates

In general, SC can easily handle computational tasks that repeat on a short interval of about a millisecond (or less, within reason). However, if these tasks include screen updates, such as GUI manipulations or `postln` calls, which may only be drawn 60 times a second, and noticed by the user even less often, we are not managing tasks efficiently. This can become a problem when using a GUI to display the state of a controller that streams data at an unusually high rate. Such a high volume of data is not necessary to create a meaningful graphical display. To this end, `SkipJack` is a utility that allows a function to be repeatedly run in the background at a regular interval, which is useful for imposing a fixed refresh rate on a GUI. The code in figure 12.20 simulates a jittery, high-density stream of data from an imaginary source (a motion sensor, perhaps). Instead of placing GUI calls inside the `~dataSim` routine, SkipJack handles the updates by setting the slider value once every 0.2 seconds. SkipJack survives pressing command period and must be stopped manually with `stop`. If you accidentally create an unnamed instance of SkipJack, it can be stopped with `SkipJack.stopAll`.

```
(
~data = 0.5;
~dataSim = Routine({
   inf.do({|n|
   ~data = sin(n * 0.01).linlin(-1,1,0.2,0.8) + rrand(-0.02, 0.02);
   (1/120).wait; // 120 data values per second
   });
}).play;
~slider = Slider().value_(~data).enabled_(false);
w = Window.new("SkipJack", Rect(100, 100, 150, 500))
.layout_(VLayout(~slider))
.onClose_({~dataSim.stop; ~sj.stop;})
.front;
~sj = SkipJack({~slider.value_(~data)}, 0.2);
)
w.close; // also stops the Routine and SkipJack
```

Figure 12.20
Using `SkipJack` to update a GUI element.

12.5 Conclusions

Like learning a musical instrument or developing a craft, practice is essential for improving proficiency and fluency with GUIs. A good first step is to study the examples in this chapter (as well as GUI code written by other experienced users), and try to build similar structures on your own. As a more substantial exercise, you can try to recreate the visual appearance of your favorite software plug-in or analog synthesizer. Perhaps the most important thing to remember is that a GUI is meant to be *functional* rather than decorative. Although it can be tempting to dress up a project with flashy colors, not every project will tangibly benefit from the addition of a GUI. A good GUI serves a clear purpose, exists in a well-defined context, follows well-established design patterns, and enhances the user experience.

Notes

1. https://doc.sccode.org/Overviews/GUI-Classes.html.
2. https://doc.sccode.org/Guides/GUI-Layout-Management.html.
3. https://doc.sccode.org/Classes/Color.html.
4. https://doc.sccode.org/Classes/SimpleNumber.html.
5. http://doc.sccode.org/Classes/View.html#Key%20and%20mouse%20event%20processing.
6. https://doc.sccode.org/Classes/Pen.html.

IV PRACTICAL APPLICATIONS

13 Sonification and Auditory Display in SuperCollider

Alberto de Campo, Julian Rohrhuber, Katharina Vogt, and Till Bovermann

13.1 Introduction

Sonification makes the inaudible audible. It perceptualizes data, formalisms, and processes through sound. Sonification is a fascinating activity from a number of perspectives: modern societies deal with large amounts of data, especially in science, but also in politics and economics. Formalisms and processes may be difficult to convey, even for specialists. Human–computer interaction (HCI) researchers are becoming more aware of the potential strengths of auditory perception as a communication channel, and many avant-garde musicians and media artists have taken up sonification as an interdisciplinary art/science venture.

For musicians, sonification is particularly interesting: data and their context provide extramusical "content" for experimental music and sound art projects. Sonification also addresses a central problem in experimental computer music in an interesting way; while designing new synthesis variants is easy, creating control data that structure them in artistically satisfying ways is not trivial. Sonification inverts this problem: given data may contain interesting structures and patterns, and inventing audible translations that are sensitive enough to make these patterns emerge perceptually is challenging. Finally, recorded data come with theories, formalizations, and models. This extends the spectrum of possible projects into simulations, mathematics, and further into the many ways to understand what really is happening in recorded data, the formalizations and theories that contextualize them, and the processes that generate them.

This chapter covers general background and central concepts of sonification and provides some smaller examples in detail, as well as a few extended examples of sonification and auditory display applications realized in SuperCollider (SC). Drawing on our experience in *artistic, scientific, and hybrid projects involving sonification*, we show how the interactive nature of SuperCollider offers flexible ways to manipulate static data, run scientific models and other processes of interest in real time, and design a variety of perceptually differentiated alternative translations into sound.

13.2 A Short History of Sonification

13.2.1 Precursors

The prehistory and early history of sonification were covered authoritatively by Kramer (1994b), and more recently Worrall (2018). Employing auditory perception for scientific research was not always as unusual as it is in today's visually dominated scientific cultures: in medicine, auscultation (listening to the body's internal sounds for diagnostic purposes) was practiced in Hippocrates's time (McKusick et al., 1957), long before the invention of the stethoscope in 1819. In engineering, some mechanics have excelled at listening to machines; for example, expert car mechanics can often tell where problems originate just by listening to a running engine (Bijsterveld 2013).

Galileo Galilei seems to have verified the quadratic law of falling bodies by listening. If strings are attached across an inclined plane at distances according to the quadratic law (1, 4, 9, 16, etc.), a ball running down the plane would produce regular sound impulses (Drake, 1975). The best chronometers of the time, water clocks, were much too imprecise to measure this, and a reconstruction of Galileo's experiment by Riess et al. (2005) demonstrated that listening for rhythmic accuracy is the most plausible procedure.

The Geiger-Müller counter renders an imperceptible environment variable audible: when radioactive decay particles hit a detector, they cause clicks, and the density of these clicks informs users about the present radiation intensity. Passive sonar, listening to underwater sound to determine the distances and directions of ships, was experimented with by Leonardo da Vinci (Kramer, 1994b); in active sonar, sound pulses are produced to penetrate visually opaque volumes of water. Listening to reflections allows for analyzing local topography and detecting moving objects like vessels, whales, or fish swarms.

In seismology, Speeth (1961) had subjects try to differentiate among seismograms of natural earthquakes and artificial explosions by listening to speeded-up recordings. Although subjects could classify the data very successfully and rapidly, little use was made of this until Hayward (1994), and later Dombois (2001), revived the discussion.

Pereverzev et al. (1997) reported auditory proof of a long-standing hypothesis: in the early 1960s, Josephson and Feynman had predicted quantum oscillations between weakly coupled reservoirs of superfluid helium; 30 years later, the effect was verified by listening to an amplified vibration sensor signal of these mass-current oscillations.

13.2.2 Sonification as a Discipline

The official history of sonification as an art/science research discipline began with the first International Conference on Auditory Display (ICAD)[1] in 1992, which brought many researchers working on related topics into one interdisciplinary research community. The conference proceedings book (Kramer, 1994a) is still a seminal reference for the field, and ICAD conferences are still central events for sonification

researchers. The community published *The Sonification Handbook*, the most current standard reference (Hermann et al., 2011), 20 years later. More recently, Worrall (2019) published a handbook on sonification design, focusing on big data sonification.

Since 1992, several research centers have been focusing on scientific applications of sonification and auditory display. The Sonification Lab at Georgia Tech led by Bruce Walker,[2] with publications dating back to 1995, also hosts the all ICAD proceedings.[3] Several universities worldwide have since established professorships for sonification and included it in their curricula.

At Bielefeld University in Germany, the Ambient Intelligence Group,[4] led by Thomas Hermann, has been contributing many interesting ideas, from theoretical concepts to creative applications, often using SuperCollider. Since the SonEnvir project[5] (2005–2007), the Institute for Electronic Music and Acoustics (IEM) at the Music University Graz, Austria, has put sonification on its research agenda and has hosted several sonification research projects,[6] applying artistic and psychoacoustic knowledge to both scientific applications and artistic projects. Also, the IEM is teaching sonification in audio engineering, sound design, and computer music curricula.

At Northumbria University in England, the Computer and Information Sciences Department has a professorship in computer science and sonification, focusing on scientific sonification, for instance the ongoing project RADICAL,[7] where current 3D audio innovations (Zotter et al., 2017) are applied to sonification.

Sonification of movement in sports or rehabilitation has been a growing area, and the Institute of Sport Science at Leibniz University in Hannover, Germany, has played a major role in exploring sonification in the context of motor learning and high-performance sports. And, last on our incomplete list, the Department of Speech, Music and Hearing at the KTH Royal Institute of Technology, Stockholm, has a long-term research focus on sonification and sonic interaction design.

Beside furthering the state of the art, the ICAD community is also very active in disseminating its knowledge, for example via the open-access *Sonification Handbook*,[8] as well as an online sonification design database.[9]

13.2.3 The Interdisciplinary Character of Sonification

The historical interplay between scientific and artistic approaches to hearing and sonification can be shown in neurology, where neurophysiologists were listening to nerve signals by telephone (Wedensky, 1883), to many scientific EEG sonifications presented at ICADs (Baier et al., 2007; Hermann et al., 2006; Hinterberger and Baier, 2005; Väljamäe, 2013), and sonifying EEG data becoming the topic of the first ICAD concert/contest in 2004.[10] Musicians' fascination with brain waves gained through exposure to Alvin Lucier's "Music for Solo Performer" (1965), analyzed by musicologists Straebel and Tholen (2014) and discussed in neurological context by

Lutters and Koehler (2017). In addition, David Rosenboom has explored many related approaches based on bio-feedback (Rosenboom 1990).

Mutual interest in the other side of this interdisciplinary domain continues to grow: traditionally research/technology-oriented venues like ICAD now consistently include curated concerts and exhibitions of artistic works, while experimental musicians and sound artists name sonification as the working method and inspiration source for their practice, especially in the context of artistic research; e.g. sound artist Scot Gresham-Lancaster currently is curating a website on sonification.[11] Experimental music and art festivals and conferences like *ctm* in Berlin[12] or *xcoax*[13] now commonly feature works involving sonification, and there have even been full festivals centered around artistic forms of sonification.[14] Large research institutions have been discovering sound and sonification for scientific outreach, such as for the detection of gravity waves by the LIGO project in 2016;[15] at CERN, experiments with sonification began in 2010 (see section 13.6).

Since antiquity, relationships between the movements of heavenly bodies and harmonicity in music have been the subject of much speculation, expressed in the concept of the music of the spheres; astronomy is still a very attractive topic for scientific outreach, and recently, NASA has been supporting a project that produces a series of audiovisual pieces as well as data sonification tutorials in Python.[16] The SuperCollider-based project Leuchtstoffraum (a cooperation between the University of Cologne and the Institute for Music and Media, Düsseldorf) explores the sonification of spectroscopy data of interstellar nebulae, showing harmonic and extremely disharmonic relations in the rotation states of free-floating molecules.

Sonification has also gained attention in sound studies (Akiyama & Sterne 2011) and science and technology studies (Supper 2012). Andreopoulou and Goudarzi (2021) studied the corpus of ICAD research and found that 14 percent of the published sonifications are artistic, compared to 79 percent scientific (and 7 percent in other areas). Furthermore, the focus shifted from music in the earlier years over design topics to interactive systems in recent years. An overview of the number of published papers on the topic sonification can be seen in figure 13.1. After years of steady but somewhat stagnant research activity, the community self-critically discussed the question of whether sonification might be doomed to fail in its claim to provide substantially new insights for various scientific disciplines (Neuhoff 2019). However, given the existing literature on the visual in science philosophy and the growing literature on the epistemology of the acoustic, sonification is likely to continue to inspire philosophy and science studies.

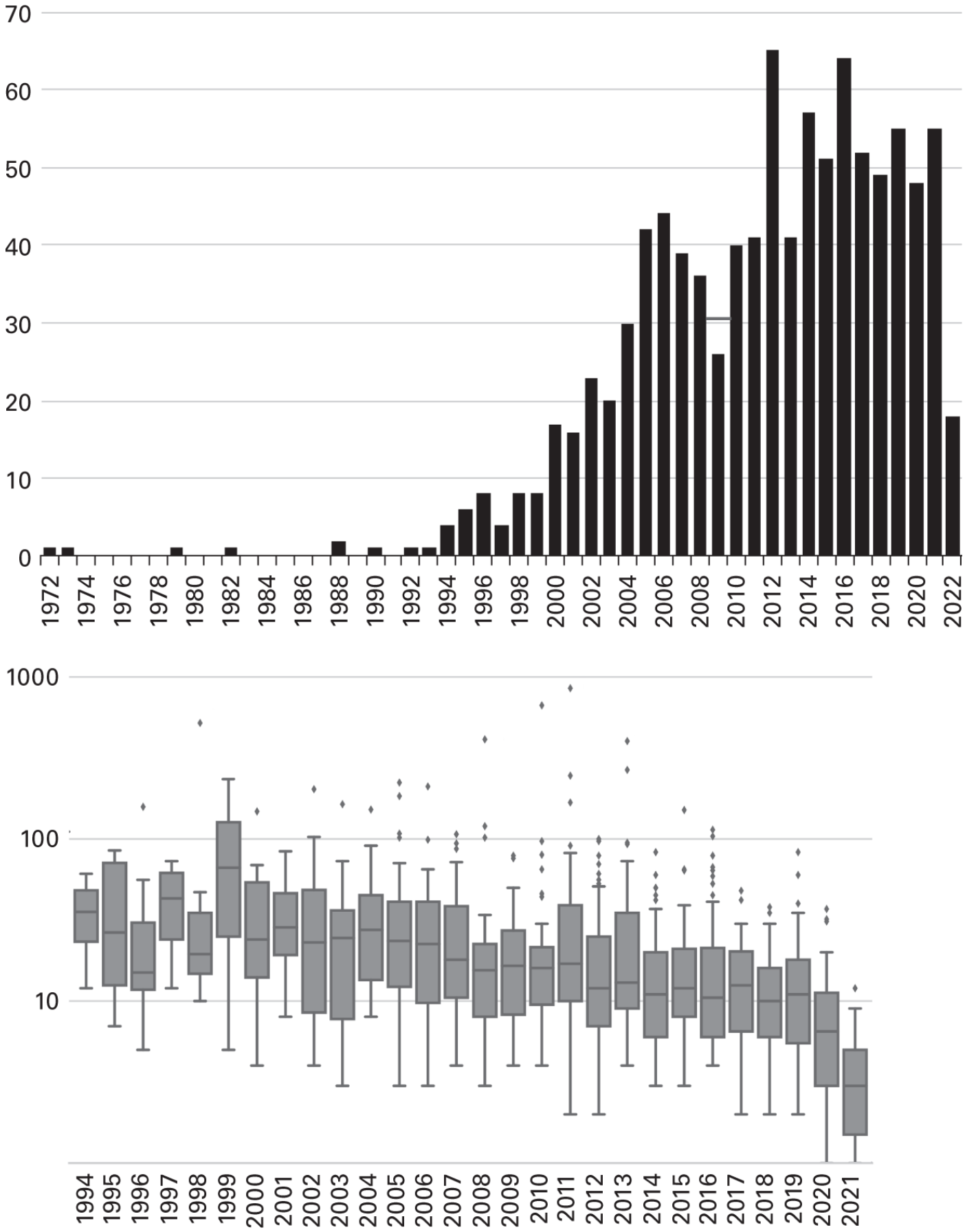


Figure 13.1
Overview of a dataset on the number of publications and their citations on the topic "sonification, auditory display,

audification, sonify" (first 1000 hits of a Google search from 2023; (Groß-Vogt et al., 2023)). (a) shows the number of papers published per year. (b) is a boxplot of the number of citations per year.

Sonification is not a smooth tool of research, and what appears to be a hindrance to general acceptance should in fact be counted as a unique opportunity—it calls for people who are ready to invest time and effort in multiple disciplines and to help build transdisciplinary organization. SuperCollider, being a programming environment rather than a computer music application, has inspired sonification and has grown communities whose interests lie between and beyond typical applications.

13.2.4 Hearing Is the Basis

When designing sonification, the central concern is to make something audible in order to gain an understanding of it. This understanding is not a relation between data points and sound signals, but between features of the data and human listeners interpreting the sound that they hear. Auditory perception is complex, consisting of the physiological responses of the auditory system, the perceptual and cognitive experience of the listeners, and, since the perception-action loop is closed, their behaviors and actions (Neuhoff 2004; Serafin et al., 2011). Traditional psychoacoustics focuses only on separated parts of the "auditory puzzle" (Neuhoff, 2004), usually lower-level sensory processes like the dependence between pitch and loudness perception or the localization of sound sources. Newer approaches explore the higher-level cognitive and ecological perspective of auditory perception in order to study listening behavior.

New sonification designs are rarely thoroughly evaluated, even though it is very useful to know how well they convey what they are intended to convey. Where full evaluation of a sonification design is not feasible, it should still be informed by insights from ecological psychoacoustics. A seminal synthesis of research in this field is Al Bregman's auditory scene analysis (ASA), a model for the capabilities of the auditory system to organize sound into perceptually meaningful elements (Bregman 1990). To understand the extraordinary capabilities of the auditory system, Bregman employs the metaphor of a lake, on which you only observe two little areas on the surface—corresponding to the acoustic vibrations at our eardrums—and from which you may infer and identify objects passing by. ASA reveals the heuristics that our cognition uses to make sense of limited information from sensory input. To show how our perception creates emergent gestalts by stream segregation and grouping, the two simultaneous processes that govern our auditory perception, Bregman provided an online collection of sound examples (Bregman, 1990b).

We want to give one example of an ASA demonstration: *Track 18: Isolation of a Frequency Component Based on Mistuning*. Usually, a specific partial within a harmonic series of overtones cannot be heard separately; all the partials and the

fundamental are perceived as fusing into one auditory gestalt. With slight detuning of this one partial, it begins to emerge and is perceptually segregated as its own, second auditory stream. In SuperCollider, this example can be easily constructed: the position of the cursor changes the detuning of one partial. On the the left side of the monitor, the partial is perfectly tuned; moving to the right, it emerges.

```
// change the frequency of one component by cursor x-position:
(
var fundamentalFreq = 300;
{
    // detuning from left 1 = in tune, to right 1.05 = 5% higher
    var detuning = MouseX.kr(1, 1.05, 'exponential');
    // partials 1-6, partial 4 is detunable
    var partials = [1, 2, 3, 4 * detuning, 5, 6];
    var spectrum = SinOsc.ar(fundamentalFreq * partials).mean * 0.1;
    // duplicate the result for stereo playback
    spectrum.dup(2)
}.play
);
```

Figure 13.2
ASA—example of detuning one partial of a harmonic sound.

While experimenting with sonification designs, it is often helpful to study how well which differentiations in sound are distinguishable in listening. Chapter 16 covers such exploration of the potential of sound synthesis processes for creating rich perceptual detail.

13.2.5 Sonification and SuperCollider

Soon after its first release in 1996, sonification projects were realized in SuperCollider, such as Mark Ballora's heart rate variability sonification (Ballora, 2000) and Julian Rohrhuber's amino acids sonification (2001, now in ProteinBioSynthesis Quark). Thomas Hermann's group at Bielefeld University[17] has conducted a wide range of research on sonification, human–computer interaction, and cognitive interaction with SuperCollider, among others, T*angible Auditory Interfaces* (Bovermann 2010). The IEM entered sonification research with the SonEnvir project,[18] where the team designed sonifications collaboratively in four scientific contexts (de Campo 2009). This project inspired a series of follow-up projects, including QCD-audio (Vogt, 2010) and SysSon[19] (Rutz et al. 2015), and initiated the Science by Ear workshop series[20] at IEM, an interdisciplinary exploration format, where domain specialists and sonification researchers explore possibilities together intensively.

The *Sonification Handbook* provides solid scientific grounding of many aspects of sonification, and includes a chapter on "Laboratory Methods for Experimental Sonification" that features SuperCollider (Bovermann et al. 2011).[21]

In recent years, several efforts have been made to provide sonification access to people with little or no sound-centered coding experience. General-purpose data sonification toolkits like the Sonification Sandbox (Walker & Cothran 2003), the Aesthetic Sonification Toolkit (Beilharz & Ferguson 2009), or the Interactive Sonification Toolkit (Pauletto & Hunt 2004) mostly failed to generate interesting sonification due to their restrictions in mapping choices, sounds, and readable data formats. Therefore, in order to lower the access barrier for scietific users with little sound background, several projects have aimed to integrate SuperCollider with widespread scientific software such as Python, where the data reading and preprocessing can be done there.[22]

13.3 Some Sonification Theory

Especially when it comes to conveying domain knowledge through sonification, scientific and artistic perspectives on sonification can conflict with each other; where scientific definitions emphasize methodological transparency and perceptual efficiency of design choices for specific practical applications, artists may focus on other aspects altogether. Mardakheh and Wilson (2022) propose an elegant resolution by identifying three strata at which artistic sonification creation processes may operate: At the *generative layer,* the data are only raw material for creating sonically interesting material to compose with. At the *allusive layer,* finding semantic connections between data context and sonic experience comes into focus, infusing the sonic understanding with domain-related meaning. Finally, at the *curatorial layer*, the goal is to accurately convey certain salient features of the data through sound to the listener in a meaningful way so that they are understood, in effect moving toward the precision and clarity demanded of sonification in scientific contexts. In interdisciplinary projects, these terms can be helpful to resolve accumulating mutual misconceptions.

Generally, such projects often reap their benefits outside the agenda. Insights may sometimes simply result from spending more time with data; they may also result from sonification being a means to cooperate below the threshold of stabilized knowledge. The allusive layer provides a setting for renegotiating tacit concepts in a conversational learning process.

Regarding the entities to be perceptualized, we find it useful to distinguish those containing well-understood information from those potentially containing unknown patterns. For well-known situations, establishing easy-to-grasp analogies is central. For exploratory situations, the task is enabling the perceptual emergence of latent

phenomena of unforeseeable types, in effect using human auditory perception as an unspecific, but potentially highly sensitive pattern detector (see section 13.2.4).

13.3.1 Sonification Concepts

For orientation in the landscape of possible approaches, we propose some working definitions of common terms in sonification research.

Auditory display is the rendering of data and/or information into sound designed for human listening. This is the most general, all-encompassing term.

Here, we differentiate between two subspecies:

Auditory information display is the rendering of well-understood information into sound designed for communication to human beings. This can include speech messages, as in airports; auditory feedback sounds on computers; and alarms and warning systems. In some definitions, the latter are also considered sonifications, specifically Earcons and Auditory Icons (see chapters 13 and 14 in Hermann et al, 2011) which are widespread as notification sounds in HCI.

Sonification, in the stricter sense, is the rendering of (typically scientific) data, formalisms, and processes into (typically nonspeech) sound, designed for human auditory perception. The informational value of the rendering is sometimes unknown beforehand, and then it is experimental/exploratory in nature.

Other definitions, especially Hermann (2008), see sonification as a purely scientific practice: “Sonification is the data-dependent generation of sound, if the transformation is systematic, objective and reproducible, so that it can be used as scientific method.” Despite such narrower definitions, just as in visualization, sonification is widespread in arts and computer music, without following these limitations. Therefore, the most accepted definition of sonification is still the earliest: “Sonification is the use of nonspeech audio to convey information” (Kramer et al., 1994a), which does not assume a particular application and covers sonification of data, formalisms, and processes equally.

We can differentiate further as follows (based on de Campo 2009 and Worrall 2019): continuous versus discrete data representations; interactive versus noninteractive sonifications; and measured data versus formal data. *Continuous representation* treats data as if they were sampled continuous signals, relying on two preconditions: equal distances along at least one dimension, typically time or space; and sufficient sampling rate. Treating data directly as time-domain audio waveforms is called *audification*, with frequency domain treatment being rare; of course, data-driven signals are also commonly used for modulating synthesis parameters, which is a form of *parameter mapping*. In audification, data points are interpreted as the instantaneous amplitude of a signal. Its playback rate is chosen so the resulting sound fits within human hearing

range. Some authors have pushed toward a purist use of audification (see chapter 12 in Hermann et al., 2011), not allowing any modulations or other DSP effects. Still, many sonifications are based on audified signals with further processing. A basic example for audifying an arbitrary function is given in figure 13.3.

```
a = [0] ++ Array.fill(1000, {|i| cos(((i+1)/10)*(sin((i/10))))});
a.plot;
b = Buffer.loadCollection(s, a, 1, {"Buffer ready".inform});
(
SynthDef(\bufferAud, {|out=0, buf = 0, rate = 0.05, amp = 0.5|
   var synthesis = PlayBuf.ar(1, buf, rate, 1, 0);
   Out.ar(out, HPF.ar(synthesis * amp, 50));
   // (Highpass filter to remove DC and protect loudspeakers)
}).add;
)
x = Synth(\bufferAud, [\rate, 0.03, \buf, b]);
x.free;
```

Figure 13.3
Basic audification of an arbitrary function

Discrete point representation creates individual sonic events for each data point or domain event, often with several dimensions of a data point being mapped to several parameters of the sound event (the classical form of *parameter mapping*, PM). Here, one can easily arrange the data in different orders, e.g., choosing subsets by certain criteria, or with a spatial layout that users can interactively navigate. Simple discrete point data representation also includes the Earcons and Auditory Icons now widely used in HCI (such as the crumpling paper sound that represents throwing a file in the trash). As a more complex application of discrete representation, PM is so widely used that it is sometimes equated with the term *sonification*. PM employs the different qualities of sound, such as pitch, loudness, rhythm, and timbre, and maps different data dimensions to these qualities. More complicated PMs are difficult to interpret by listening, as the psychoacoustic entities used are nonorthogonal. The most common PM is the Auditory Graph (Flowers, 2005; Nees, 2018), where one-dimensional data points of a 2D graph are mapped to time (*x*-axis) and pitch (*y*-axis) with discrete sounds (often, simple sine beeps).

```
// The simplest auditory graph: beeps with pitch varying over time
d = [1.1, 5, 9, 3, 7, 9, 11, 42]; // an arbitrary data set
// map the data logarithmically into a well-heard frequency range
n = d.linexp(d.minItem, d.maxItem, 300, 500);
```

```
(
SynthDef(\SimpleSine, {|freq = 440|
   Out.ar(0,
        SinOsc.ar(freq)
        * EnvGen.ar(Env.perc(0.1, 0.8, 1, -4), doneAction: 2)   *
 0.2
   )
}).add;
)
(
Task({
 n.do({arg i;
   i.postln;
        Synth(\SimpleSine, [\freq, i]);
   0.3.wait;
});
}).play;
)
```

Figure 13.4
A basic auditory graph.

An essential design dimension is the degree of *interaction* that a sonification provides. In music, we commonly assume that learning a piece or concept is much deeper when trying to play it as opposed to only listening to it; this makes properly *interactive sonification* approaches attractive and promising. For instance, *model-based sonification* (discussed in detail in chapter 16 of Hermann et al., 2011) employs a more complex mediation between data and sound by introducing an acoustic model whose properties are informed by the data. Such models are well suited to capturing domain knowledge, and they transfer well to different application domains. As any acoustic model, they need an excitation and thus are inherently interactive.

```
// We employ FreeVerb as sound model, whose parameters are set by t
he position of the mouse (i.e., our data)
// Use headphones!
{FreeVerb.ar(SoundIn.ar([0, 1]),
   MouseX.kr(0.0, 1.0).poll, MouseY.kr(0.0, 1.0).poll, 0.5)
}.play;
```

Figure 13.5
Basic example of model-based sonification.

All these concepts are based on the idea that a sound-generating process is parameterized only by measured data from the domain. Conversely, *operator-based sonification* (Rohrhuber 2010, Vogt and Höldrich 2011) is based on formal data. It shifts the emphasis to a domain formalism (such as a physical law or a domain model) that is made audible by mapping it onto the sonification time domain. This process can then in turn be informed by data and may employ any of the other methods. Operator-based representation treats data and theoretical context on equal footing. For this, domain formalisms are made part of the sonic representation. Any empirical variables that represent the data become audible through the formal relationships between them, which is particularly useful when one is interested in structures that can be expressed in mathematical terms. The method gives a systematic perspective on a wide range of sonifications and can be made precise in so-called *mixed expressions* (see section 13.3.2).

The usual concept of "data sonification" turns out as one extreme, where all the formalism represents the sound synthesis algorithm and the parameters are informed by the data. On the other end of the spectrum, we find cases where the formalism represents the structures inherent in the domain without any data to further inform it (for examples, see section 13.5.2). In moving between these extremes, sonifications can contribute to the scientific task to mediate between the theoretical and the empirical.

Which concept is most adequate for a given research problem is usually not clear initially. Using a sound programming language in the collaborative process provides a way to *navigate* possibilities rather than trying to merely implement a preconceived imagination about its result. Among the many options to enable such a navigation, we would like to briefly mention a few that have been developed in the sonification community.

13.3.2 Tools to Facilitate Sonification Design

Throughout ICAD's history, several attempts have been made to establish design criteria, from using a task and data analysis questionnaire (Barrass 1997) to a recent contribution of Nees (2019), transferring the requirements of established design theory to sonification. Enge et al. (2021, 2023) recently proposed to link sonification theory closer to the more standardized theory of visualization. While the substrate of a visualization is the map, this corresponds to time in sonification. A zero-dimensional mark (i.e., a visual point) corresponds to one static auditory gestalt. This can be a simple sine beep, or any other sound event of variable duration that does not change in time. A 1D-mark (i.e., a visual line) corresponds to an evolving sound. The different channels (such as colors) in visualization then correspond to the duration, timbre, pitch, etc., of the sound. While a general framework for sonification design is still out of reach, various contributions are already helpful for sonification practice.

The Sonification Design Space Map (de Campo, 2007, 2009) puts the above forms of representation in one context. It suggests preferred sonification strategies to experiment with, based on general properties (number of data points, number of data dimensions), current working hypotheses (which data subset sizes are expected to contain patterns), and perceptual considerations (most important, sonifying parts of the domain within time frames suitable for human working memory). Within this conceptual framework, the map supports reasoning about the next experimental design steps. Successive design choices can be understood and communicated as movements on the map.

The use of *mixed expressions* (Rohrhuber 2010, Vogt and Höldrich 2011) is a way to bring sonification into the research domain by augmenting the standard style of formal expression with expressions from sound synthesis. This avoids a predefined division of labor and allows participants to understand more complex sonifications. A sonification operator formalizes a concise link between a domain science (any) and an algorithm of sound synthesis: the sonification signal depends on the continuous listening time t, data d, and parameters p of the sonification. Examples for mixed expressions can be found in Vogt (2010).

$$\overset{\circ}{S} : \mathcal{A}(d) \rightarrow \overset{\circ}{y}(\overset{\circ}{t};d,p)$$

Incorporating methods from *user-centered design* led to including domain scientists from the beginning of the sonification design process (Goudarzi et al., 2013). The elaboration of metaphors from domain science and their translation into the audio domain may help for better acceptance within the specific communities (Höldrich & Vogt, 2010).

Evaluation of the developed prototypes should constitute a central aspect of the sonification process, but unfortunately it is rarely realized. General methods for evaluating sound-producing objects can be applied to sonification as well (Giordano et al., 2013). Profound evaluations of sonifications usually focus on their functionality (for instance, in guidance tasks (Parseihian, 2016) or physiotherapy (Vogt et al., 2010a)), and only rarely on their aesthetics (Vogt et al., 2013). The necessity of dealing with sonification from an aesthetic point of view has been discussed by Vickers et al. (2017), exploring sonification aesthetics that are based on the experience of a subject, while this experience is being constrained by the scope of what the sonification designer intents to reveal.

Any list of conceptual and practical tools for the sonification design process is necessarily subjective and incomplete, as researchers continuously develop new approaches and methods. We discuss the most universal ones in the following section.

13.4 Basic Considerations for Sonification Design

The two central concerns in sonification design are *mapping,* i.e., which data or process properties should control which aspects of the sounds created, and *time scaling* (usually how many data points should be rendered within a suitable time frame, so patterns in the data become clearly noticeable auditory gestalts).

13.4.1 Mapping

Mapping data, formalisms, or processes onto perceptible aspects of sound requires some familiarity with auditory sensations and their physical correlates. We consider a few prominent cases next.

Pitch—frequency: The human ear resolves pitch quite well; in the middle range, one can discern frequency modulations of less than 0.5 percent, and for individual tones, about 1 percent (or 1/6 of a semitone) can be distinguished. Such a threshold is called the "just-noticeable difference (JND)." Extrapolated to a range of 100–6,000 Hz, this allows about 400 distinguishable steps.

Loudness—amplitude: For amplitude, the JND is between 0.5 and 1 dB for most sounds, so given a usable dynamic range of 50 dB, this is roughly 50 resolvable steps. Because listeners always adjust to slowly changing loudness levels, amplitude is not a good choice for rendering high-resolution data. However, one can use amplitude in simultaneous sounds to convey relative importance: softer sounds draw less attention in a complex sound scene, placing them in the background.

Localization—spatial position: Although humans can detect changes in direction of a physical sound source on the order of a few degrees, panning as implemented on most audio systems relies on phantom sources created by energy distribution across speakers; outside the sweet spot, this illusion becomes fragile. Thus, relying on fine panning resolution is not recommended. However, for distinguishing parallel auditory streams, panning is very efficient, especially when every source has an independent loudspeaker. Then listeners can rebalance the overall soundscape mix by moving closer to sources of interest.

Timbre: Although there is no general definition of timbre, most other properties are subsumed under it, and some can be used in sonification. For instance, brightness works as a perceptual concept for most listeners, where auditory resolution will depend on the particular synthesis algorithm. One can always programmatically generate examples of the range intended to be used and try to verify plausible ranges of sonic variation. Because timbre is a complex and underspecified property, it is something to *listen for* in search for unknown information about the domain.

Many dimensions interact; for example, amplitude and frequency: sine tones below 200 Hz and above about 10 kHz will appear progressively softer, while sine tones

between 2 and 4 kHz will appear louder. These curves, first measured by Fletcher and Munson (1933), have been the basis for many standards for equal loudness measurement. In SC3, the `AmpComp` and `AmpCompA` classes provide compensation for this dependency.

Understanding meaning in sonification depends crucially on the metaphors implied. Consider rendering measured temperature: higher temperatures may be mapped to higher pitch, higher event density, brighter timbre, etc; both polarity and scaling of numerical ranges (Walker, 2002), as well as choices for other mapped parameters, will influence how easily a sonification design is understood by its intended listeners. When it is not known what property will eventually become relevant, it may be useful to keep the metaphors fluid, or if possible evaluate the mapping in basic listening experiments.

13.4.2 Time Scaling

To find patterns of interest, one needs a working hypothesis on what aspect of the data, formalism, or process may contain them. Patterns may occur at different scales, such as near-repetitions of groups of tens of data points or slow trends over millions of data points. A single sonification design is unlikely to render both as easily perceptible gestalts, but designs with flexible time scaling may well work adaptively for a wide range of data subset sizes and ordering choices. Perceptual aspects to consider here are event fusion (below 30-ms time distance, individual auditory events fuse; above that, they become separately perceivable), and the memory limit of a few seconds within which sonic phrases can be kept in vivid, raw detail (i.e., before perceptual categorization processes simplify the experience).

As an example to show the dependence of time scaling and pitch mapping, consider focused audification (Groß-Vogt et al., 2020). This concept allows for switching between pure audification and continuous auditory graph (mapping data values to pitch of a continuous, interpolated, sound). When focusing on a data set (i.e., extending the time scaling by setting a smaller sampling rate of the input signal), the pitch of the resulting audification is lowered. To compensate for this, a Hilbert transform is used as a frequency shift of the factor `deltaf` and allows one to reposition the sound in a well-heard frequency range. By this transposition, the differences between data points are affected as well, as in a higher frequency range, the difference between absolute frequencies perceptually appears closer than in a lower range. Thus, a further parameter makes it possible to manipulate this factor, which is called the constant `c`. For setting `rate=1`, `deltaf=0` and `c=0`, the algorithm outputs a pure audification at standard sampling rate. For any data set, the optimal listening range can be evaluated, setting the sampling rate, `deltaf`, and `c`, possibly also exploring the data set interactively.[23]

```
// This dummy data set simulates a rising climate signal over a per
iod of 30 years with 12 monthly values for mean temperature each
a=Array.fill(30*12, {|i| sin((i.rrand(i+3))/2) + ((i-10).abs.   rra
nd(i+10)+1/200)});
b = Buffer.loadCollection(s, a, 1);
b.plot;
(
SynthDef(\fa, {
   |rate = 0.05, c = 0.3, deltaf = 0, b = 0, startpos = 0|
   var fMod, sin, cos, hilb, sig;
   sig = PlayBuf.ar(
        1, b, rate: BufRateScale.kr(b) *rate,
        startPos:startpos, loop: 1);
   fMod = deltaf * (2 ** (sig * c));
   sin = SinOsc.ar(fMod);
   cos = SinOsc.ar(fMod, 0.5pi);
   hilb = HilbertFIR.ar(sig, LocalBuf(4096)) * [cos, sin];
   Out.ar(0, hilb[0]-hilb[1]!2 * 0.1);
}).add;
)

x = Synth.new(\fa, [\b, b, \c, 0, \deltaf, 0]);
x.set(\rate, 0.01);
x.set(\deltaf, 500);
x.set(\c, 0.3);
x.free;
```

Figure 13.6
Focused audification with an arbitrary example data set.

As an extreme case of time scaling, we consider *SynopsisArabidopsis,* an audiovisual installation by Alberto de Campo and Kathrin Hunze about the genome of *Arabidopsis thalliana*, the most studied plant in biology. When setting the time scaling to render individual DNA items as a fast melody (8 notes per second), rendering the entire genome takes six months—which was the duration of the exhibition. Knowing this gives the audience an idea of data dimensions, and the relative scarcity of computer-spoken annotations when DNA sections with known properties are traversed shows how much of the genome is not understood yet.[24]

13.5 Tour d'Horizon of Sonification

In this section, we present a selection of sonifications that exemplify the basic concepts and design considerations discussed above, and show how differently they can play out in a wide variety of sonification.

13.5.1 Human Rights—an Example

To illustrate the sonification strategies and mapping considerations discussed above, we reconsider experiments with sonifying social data in 1998 (de Campo and Egger de Campo, 1999); it shows sonification variants with a simple data set concerning an important human rights issue. The data are social statistics[25] (namely, the number of death penalties carried out in the United States since reinstatement of the death penalty in 1977 up to 2007, split into four regions: Northeast, West, Midwest, and South). We include data for the entire United States and separately for the state of Texas. In figure 13.7, we load the data into memory.

```
(
q = q? ();
q.execdata = ();
q.execdata.years = (1977 .. 2008);
   // data is: [total for each region, 1977. . .2007];
q.execdata.regions = (
   Total: [1099, 1,0,2,0,1, 2,5,21, 18,18, 25,11,16,23,14,
        31,38,31,56,45, 74,68,98,85,66, 71,65,59,60,53,42,37],
   Northeast: [4, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,
        0,0,0,2,0, 0,0,1,0,0, 0,0,0,1,0,0,0],
   Midwest:   [129, 0,0,0,0,1, 0,0,0,1,0, 0,0,1,5,1,
        1,4,3,11,9, 10,5,12,5,10, 9,7,7,14,6,5,2],
   South: [933, 0,0,1,0,0, 2,5,21,16,18, 24,10,13,17,13,
        26,30,26,41,29, 60,55,74,76,50, 61,57,50,43,44,36,35],
   West:[67, 1,0,1,0,0, 0,0,0,1,0, 1,1,2,1,0,
        4,4,2,2,7, 4,8,11,4,4, 1,0,2,2,3,1,0],
   Texas: [423, 0,0,0,0,0, 1,0,3,6,10, 6,3,4,4,5,
        12,17,14,19,3, 37,20,35,40,17, 33,24,23,19,24,26,18]
);
q.getReg = {|q, regName| q.execdata.regions[regName].drop(1)};
)
```

Figure 13.7
Loading the data.

First, we represent the data points as discrete events. This design maps data values of 0 to 76 onto MIDI notes between 36 and 112, into a total time of 5 seconds (see figure 13.8).

```
(
Pbindef(\exec,
   \note, Pseq(q.getReg(\Midwest)),
   \octave, 3,
   // play entire time range in 5 seconds
   \dur, 5 / q.execdata.years.size
).play;
)
   // switch between different regions
Pbindef(\exec, \note, Pseq(q.getReg(\Northeast)));
Pbindef(\exec, \note, Pseq(q.getReg(\West)));
Pbindef(\exec, \note, Pseq(q.getReg(\South)));
```

Figure 13.8
Mapping the data to pitch in discrete events.

The South region does sound very different from the others.

Next, we try continuous sonification. This design maps data values of 0–76 to frequencies from 200 to 4,000 Hz. Note that data value interpolation does not seem very suitable here (see figure 13.9).

```
b = Buffer.sendCollection(s, q.getReg(\Northeast), 1);
(
Ndef(\exec, {|dur = 5, scale=50, offset=200|
   var values = PlayBuf.ar(1, b, dur / SampleRate.ir);
   Pan2.ar(
        SinOsc.ar(values * scale + offset), 0,
        EnvGen.kr(Env.linen(0.01, dur, 0.01, 0.2), doneAction: 2)
   );
}).play;
)
   // switch between different regions
b.sendCollection(q.getReg(\Midwest)); Ndef(\exec).send;
b.sendCollection(q.getReg(\West)); Ndef(\exec).send;
b.sendCollection(q.getReg(\South)); Ndef(\exec).send;
```

Figure 13.9
Continuous data sonification.

In the original discussions on sonifying these data, we reached agreement that, for example, a value of 5 should be represented by 5 countable sound events, and a value of 0 by silence. We also agreed on providing a ticking sound to indicate the beginning of each year. Third, panning the sound to 4 speakers should be possible for playing

multiple regions in parallel, so each region has its own sound location The final design scaled one year to 2 seconds, resulting in a total presentation time of about 1 minute, and maps data values to pulses per period (see figure 13.10).

```
(
SynthDef(\noisepulses, {|out = 0, sustain=1.0, numPulses = 0,   pan
= 0.0, amp = 0.2|
   Out.ar(out,
        PanAz.ar(4,
              PinkNoise.ar
              * Decay2.ar(Impulse.ar(numPulses / sustain, 0,   numP
ulses.sign), 0.001, 0.2),
              pan,
              EnvGen.kr(Env.linen(0.0, 0.995, 0.0), levelScale:   a
mp, timeScale: sustain, doneAction: 2)
        )
   );
}).add;

SynthDef(\tick, {|out, amp = 0.2, pan|
   OffsetOut.ar(out, Pan2.ar(Impulse.ar(0) * Line.kr(amp, amp,   0.
001, doneAction: 2), pan))
}).add;
)
(instrument: \noisepulses, numPulses: 10, legato: 1, dur: 2).play;
(instrument: \tick).play;
```

Figure 13.10
Sound design with noise pulses.

The four regions are presented sequentially as in figure 13.11, and in parallel in the book code repository.

```
(
Tdef(\execs, {

   var yearDur = 2;  // one year is 2 seconds
   var region, numExecs, numyears = q.execdata.years.size;
   // ordered by ascending total number of executions:
   [\Northeast, \Midwest, \West, \South].do {|regName, i|

        region = q.execdata.regions[regName].postln;
        q.execdata.years.do {|year, i|
```

```
                numExecs = region[i + 1];
                [regName, year, numExecs].postln;
                (instrument: \tick).play;
                if (numExecs > 0) {
                    (instrument: \noisepulses, legato: 1,
                    numPulses: numExecs, dur: yearDur).play;
                };
                yearDur.wait;
            };
            yearDur.wait;
        };
    }).play;
    )
```

Figure 13.11
Four regions in sequence.

Since 2007, many countries have explicitly ended the death penalty, while others have increased its use. Many human rights organizations monitor and analyze these worldwide developments closely, see, e.g., https://deathpenaltyinfo.org/.

13.5.2 Sonifying Formalisms

What is a sonification and what is not may depend on the perspective of those who employ it. The reason for this is simple: in every sound, we may listen to something that is not audible by itself but is rendered audible. Something as simple as

```
{SinOsc.ar([MouseX.kr(200, 3000, 1), MouseY.kr(200, 3000, 1)])   *
 0.01} .play; can be considered a minimalist, duophonic   instrumen
t. But when it is surreptitiously sent to play   on other musician
s' computers in a networked live coding   performance,26 it becomes
a virus, softly sonifying their   mouse movements.
```

Chaos UGens such as QuadN, HenonL, and GbManC employ iterative functions and differential equations with chaotic behaviors for generating sound. These could be considered audifications of the data sequence that is generated by these functions:

```
{HenonC.ar(2000, LFNoise2.kr([1, 1], 0.2, 1.4), 0.14) * 0.2}   .pla
y;
```

This demonstrates that it can be misleading to associate sonification with measured data. Its domain can also be mathematical. For example, one can sonify the relation of divisibility between numbers and derive an audible spectrum of ratios:

```
f = {|num|
   num.factors.powerset.as(Set).as(Array).collect(_.product)
   .sort.drop(1).drop(−1)
};
f.value(112233);
```

Without being explicitly called "sonification," formalisms show interesting properties and have been used in many ventures between mathematics and musical composition. The line above comes from the composition "Politiken der Frequenz" by Marcus Schmickler and Julian Rohrhuber, which is an elaboration on different mathematical concepts, such as prime factorization, Dedekind cuts, complex and surreal numbers against the backdrop of political economy and philosophy.

It is often undecidable under what conditions deterministic processes may be unpredictable, which makes such systems a good topic for algorithmic sound synthesis. Sturmian sequences, featured in compositions by Tom Johnson and Jean-Paul Allouche, are good examples of this; the piece *Sturmian Constellation* by Julian Rohrhuber is available on the book code repository. The sonification of Emil Post's *Tag Systems*[27] shows intractable behavior for some initial conditions—in the case of universal machines, these conditions may be inherently undecidable (De Mol, 2007); since the human ear is very sensitive to subtle varieties of repetition, a sonification of such algorithms reveals characteristic patterns very quickly (see figure 13.12).

```
(
// requires TagSystemUGens from https://github.com/supercollider/sc
3-plugins
// compare two axioms on left and right channels
// μ = 4 (size of alphabet)
// v (deletion number) varies [1..6] with horizontal cursor positio
n
{
   var tag, rules, val;
   rules = [[0, 1, 1], [1, 3, 2, 0], [1, 2], [3, 1, 1]];   // same
 rule for both
   v = MouseX.kr(1, 6);
   val = dup {
        var axiom = Array.fill(14, {#[0, 1, 2, 3].choose});   axio
m.join.postln;
        Duty.ar(1 / SampleRate.ir, 0, Dtag(7e5, v, axiom,   rules),
doneAction: 2);
   } * 0.1
```

```
}.play;
)
```

Figure 13.12
Sonification of tag systems.

Formal systems like these are domains where sonification lets us explore complex consequences of simple formal rules, and where the object of measurement is not external to the device being measured, at least not in the strict sense. Operator-based sonification, mentioned among the strategies above, gives us a method to "derive" sonifications from expressions in both a free and precise way. In the context of formal treatments of physical systems, it reminds us of the fact that what we hear in mechanical vibrations is not mere data points, but the internal relationships between processes and their external relation to us.

13.5.3 Boundaries of Sonification

As with most things, sonification has more than one boundary, and they shift. For instance, one could argue that algorithmic music is a sonification of time (Rohrhuber 2018). To conclude this section, we pick two other limit cases—one technical, one linguistic. Sonification of some sort features in many technical devices and helps to reveal their inner life; even the sound of a light switch indicates that something has changed. As truck drivers have a fine ear for motor and transmission, operators of telephone switching stations used to listen to the sound patterns of the mechanical relays. This practice was transposed to an early computer sonification method: a loudspeaker attached to a memory location in the processing unit helped computer operators to notice when a program had gone into a loop state and had to be externally halted (Rohrhuber 2018). Today, processor frequencies are much higher, and the electromagnetic auscultation techniques of artists like Christina Kubisch[28] and Valentina Vuksic[29] follow this call.

The technique of making audible the invisible internal state of a calculation resonates with the idea of expressing thoughts through action or speech. The composition "Mind Reader," by Marcus Schmickler and Julian Rohrhuber (2018), features the internal state of an emulation of Claude Shannon's "Mind Reading (?) Machine" (1953), which plays "odds and evens" with the machine operator. The transitions of the internal states are displayed by two sequences of ever-rising chords, which move either by the pure interval 4/3 (prediction of losing) or 5/6 (prediction of winning). Because the two intervals consist of coprimes, their iteration drifts farther and farther from each other, while the whole sound is kept in the same frequency range by smoothly transposing it in Shepard-tone octaves. This piece is a boundary case of sonification, as the object it

represents is not an internal machine state, but the Cold War theme of cognitive agents pitched against each other, without common ground and still caught in one arena.

Many definitions limit sonification to "nonspeech audio," so here is an example that questions this constraint. The radio art piece "Wandering Lake—an Atomic Opera" by Echo Ho und Ulrike Janssen[30] is about the mystical dried-up salt lake Lop Nur in Inner Mongolia, which the Chinese government decided to use as testing ground for atomic weapons. For this piece, Alberto de Campo sonified the timeline of nearly 2000 atomic bombs exploded between 1945 and 1998, using recorded and synthetic speech, for time-grid orientation and for enunciating the code names for all these bombs; they inadvertently seem expressive of the political, military, and technoscientific cultures underlying the race for atomic weapon technology (see also Volmar 2014).

13.6 Supercolliders

Our favorite programming language is named after a failed big science project in Texas, the Superconducting Super Collider (SSC), which had been planned and partially constructed for many years before being canceled in 1993. The computer science department in Austin, where James McCartney studied, was adjacent to the physics department that had to cope with the loss. The computer language SuperCollider has been repeatedly haunted by this past, attracting data and people from high-energy physics.

One of the early physics-related sonification projects started in 2008 in Hamburg; we tried to help solve to interpret collisions between protons and electrons or positrons that had been measured in HERA, the Hadron-Electron Ring Accelerator (Kowalski et al. 2010). While the work was inspiring for all participants and informed a paper on the philosophy of science (Rohrhuber 2011), it did not contribute to a solution to the intricate problem.

At CERN, the European Organization for Nuclear Research, various approaches have been taken to sonify data from different experiments. Many of them are musifications of experimental data, i.e., more traditional musical patterns of harmony and rhythm are employed to convey data (e.g., see Cherston et al., 2016). Following a more abstract, but metaphoric sonification approach, Vogt et al. (2010b) sonified data from the particle detection chamber, sonically rewinding the experiment to show the mechanism of the measurement device.[31]

The LHCSound group initiated by Lily Asquith[32] in 2010 began to explore sonifying data from the ATLAS experiment and later pursued collaborations with choreographer Gilles Jobin and composer Carla Scaletti[33]. The Dark Matter project by the Birmingham Ensemble for Electroacoustic Research (BEER) (Vasilakos et al, 2020) explored the possibilities of using LHC data as material for group live coding performances. The

data stem from collision experiments at the Large Hadron Collider at CERN, where traces of particles called “muons” and “jets” are created. The measured properties for each of these particles is recorded in an event database, and Dark Matter uses an example selection of these. Scott Wilson has kindly provided code examples for how BEER approaches live coding sonifications. Figure 13.13 shows typical live coding usage, and a translation to plain SC is in the book code repository.

```
// a synthdef to sonify one event
(
SynthDef(\ring, {arg out=0, freq, decay = 1, amp = 1;
   var ring;
        ring = Ringz.ar(Decay.ar(Impulse.ar(0), 0.03, ClipNoise.
 ar(0.01)), freq, decay);
        DetectSilence.ar(ring, doneAction:2);
        Out.ar(out, Pan2.ar(ring * amp, Rand(-1, 1)));
}).add;
)

// Custom Dark Matter pattern classes make use of the concept of a
 currently active event, which can be swapped while playing. This i
s managed behind the scenes, as is optional scaling of values to be
between 0 and 1 for easy parameter mapping. Within a Dark Matter pe
rformance, usage looks like this:
(
// step through and sonify each subatomic constituent in the curren
tly active collision event
// eta -> duration
// pt -> freq
// phi -> decay
// m -> amp
Pbind(\instrument, \ring,
   \dur, PallConstituentsS(Pseries(), "eta").linlin(0, 1, 0.1,   0.
5),
   \freq, PallConstituentsS(Pseries(), "pt").linexp(0, 1, 200,   40
0),
   \decay, PallConstituentsS(Pseries(), "phi").linlin(0, 1,   0.5,
 3),
   \amp, PallConstituentsS(Pseries(), "m").linlin(0, 1, 1, 2)
).play
)
```

Figure 13.13
Example for Dark Matter BEER approach.

SuperCollider has indeed been meeting super colliders.

13.7 Sonification Concert Pieces

We conclude this chapter with descriptions of three early concert pieces realized with SuperCollider. They prepare the ground for the intermediate space shared by aesthetic, scientific, philosophical, and historical considerations.

13.7.1 Bonner Durchmusterung

In 2008, Marcus Schmickler and Alberto de Campo created sonifications for "Bonner Durchmusterung," an audiovisual composition[34] named after a large-scale historical charting of the visible stars conducted from 1846–1863 in Bonn, Germany.[35] Based on data from different astronomical phenomena, such as solar flares, pulsars, gamma ray bursts, and gravitation models, we created scientifically plausible sonification patches that were also playable, thus working at the *curatorial* level (Mardakheh & Wilson 2022). Playing evolving versions of these patches and adding further sonic elements for transitions, Marcus created a large variety of sound material for each episode, in effect composing with sonification-based material at the allusive layer.

For this chapter, we chose gravitation models as they can run in SuperCollider with accessible parameters and generate the data informing the sound world in real time. The following code example is a minimal reimplementation of this idea: Figure 13.14 shows a gravitational model adapted from redFrik's redUniverse quark examples;[36] for a simple visualization, see the book code repository.

```
(
// make a world—requires redUniverse quark,
// based on example attraction03.scd
q = q?();
q.world = RedWorld3(RedVector[600, 600]);
q.center = q.world.dim * 0.5;
// make world objects
20.do {
   var loc = q.world.dim.rand;
   var mass = exprand(0.1, 10);
   //world, loc, vel, accel, mass, size
   RedObject(q.world, loc, 0, RedVector2D[1, 1], mass, (mass*5).
 sqrt);
};

// make world parameters tunable:
Spec.add(\gravity, [0, 2, \amp]);
```

```
Spec.add(\maxVel, [2, 200, \exp]);
Spec.add(\damping, [0, 1, \amp]);

// set world Tdef parameters
Tdef(\gravity).set(\gravity, 0.5);
Tdef(\gravity).set(\maxVel, 20);
Tdef(\gravity).set(\damping, 0.25);

// make Tdef that runs the world:
Tdef(\gravity, {|envir|
   var world = q.world;
   loop {
        // update world parameters
        world.gravity.put(1, envir.gravity);
        world.maxVel = envir.maxVel;
        world.damping = envir.damping;

        // update world dynamics of interacting objects
        world.objects.do{|o, i|
              world.objects.do{|oo, j|
                  // if not too close (to avoid very high forces)
                  if(i!=j and: {o.loc.distance(oo.loc)>(o.size*2)})
{
                  // add forces for each pair of objects
                        o.addForce(o.gravityForce(oo));
                  };
              };
              o.update;
              world.contain(o);
        };

        q.mapSynths; // map to sonifying synths
        0.0333.wait;
   }
});
);
```

Figure 13.14
Gravitational model.

Figure 13.15 shows the `SynthDef` used for the sounds that represent each object, and `q.mapSynths` defines how primary and derived object properties, such as location and speed, are mapped to the synth process parameters.

```
// make a synthdef
SynthDef('sinFB', {|out, gate = 1, amp = 0, pan, freq = 260,   fb =
2|
   var snd = SinOscFB.ar(freq.lag(0.1), fb.lag(0.1));
   var ampComp = AmpComp.kr(freq.lag(0.1));
   var env = Env.asr(2).kr(2, gate) * ampComp;
   Out.ar(out, Pan2.ar(snd, pan.lag(0.1), amp.lag(0.1) * env));
}).add;

// make a synth for each object:
q.synths = q.world.objects.collect {Synth('sinFB', [\amp, 0])};

// map the object's properties to its sound synth parameters:
q.mapSynths = {
   q.world.objects.do {|obj, i|
        //***** calculate more object properties:
        // speed, roughly normalized for maxVel 50
        var speed = obj.vel.mag / 50;
        // kinetic energy
        var energy = speed.squared * obj.mass;
        // distance from world center
        var distFromCenter = obj.loc.distance(q.center);

        //***** map the object properties to synth parameters:
        // higher energy and closer to center -> louder
        var amp = energy.sqrt * (50 / (distFromCenter + 50));
        // object x-coord -> pan
        var pan = obj.loc[0].linlin(0, 600, -1, 1);
        // use object y-coord -> pitch
        var freq = obj.loc[1].linexp(0, 600, 2000, 20);
        // higher energy means brighter sound
        var fb = energy.sqrt.linlin(0, 0.5, 0.5, 3);
        // set synth params:
        q.synths[i].set(*[\freq, freq, \amp, amp, \pan, pan,   \fb,
fb]);
   };
};
```

Figure 13.15
Sonify world objects and forces.

The following two pieces were created for the ICAD 2006 concert "Global Sound—the World by Ear."[37] The concert call invited sonifications of the social data of 190 nations, with 48 data dimensions ranging from such basic statistics as state population and area to GDP (gross domestic product) per capita to the percentage of population with good access to water and sanitation. While the underlying data are the same, the approaches taken in these pieces are radically different; as both were written in SuperCollider, they can be both discussed conceptually and shown practically.

13.7.2 Navegar é preciso, viver não é preciso

Navigation has exerted a major influence on world history; drastic changes began with expeditions by Bartolomeu Díaz, Christopher Columbus, and Vasco da Gama.[38] Ferdinand Magellan's voyage was reported by Antonio Pigafetta (1530, 1525), and novelists (e.g., Zweig, 1938) and historians have retold, idealized, and critically scrutinized the not only glorious story—as progress for Western science and cartography, irrefutably proving that the world is indeed a globe; and as an early step in European imperial ambitions, preparing the horrors of colonialism.

For "Navegar" (de Campo and Dayé, 2006), we combined the given geographical data with historically significant time/space coordinates: the route of Magellan's expedition to the Moluccan Islands (1519–1522), the first circumnavigation of the globe. In August 1519, Magellan departed for the Spice Islands with five ships and 270 crew members. They looked for a passage to the Pacific near today's Argentina, where winter storms forced them to wait for six months. One ship sank and another deserted, but three ships found the passage in October 1520 and headed across the Pacific, toward the Philippines. They landed on several islands there, and on Mactan, Magellan was killed by natives.

The crew (now down to 115) and two ships continued. They reached the Spice Islands in November 1521. Only the *Victoria* sailed across the Indian Ocean, rounding southern Africa and reaching Spain in September 1522 with 18 surviving crew and 26 tons of spices like cloves and cinnamon. One is reminded of Caetano Veloso, who, pondering the mentality and fate of the Argonauts, quoted an ancient saying in a song: "Navegar é preciso, viver não é preciso" (Seafaring is necessary, living is not). (See figure 13.16.)

Pondering which dimensions to sonify, we felt that explorers today would consider economic power and natural resources. For economic power, GDP is a myopic measure, as income is never evenly distributed. The *Gini index* measures this inequality: 0 means equal income for everyone, and 100 is all income for one person. So we kept GDP and added the income ratios of the top and bottom 10 percent and the top and bottom 20 percent, as dimensions to sonify. (In Denmark, with the lowest Gini

index of 124 nations listed, the top 10 percent earn 4.5 times more than the bottom 10 percent; in Namibia, with the lowest ranking, the ratio is 128.8:1.)

The ICAD06 Concert data set also provided the percentage of the population with adequate access to drinking water for each country. Unfortunately, this United Nations indicator has many missing values, so we substituted data values of neighboring countries, assuming similarity.

Regarding mapping choices, we deliberately chose higher display complexity, though it demands more listener concentration and attention, hoping that a more complex piece will invite repeated listenings. Every country is represented by a sound stream composed of five resonators; all parameters of this stream are determined by data properties of the associated country and the navigation process (the ship's distance and direction toward a given country). At any time, the 15 countries nearest the route are heard, which provides display clarity and reduces CPU load (see table 13.1).

Table 13.1
Mappings of Data to Sonification Parameters in "Navegar"

Data Dimension	Sonification Parameter
GDP per capita of country	Central frequency of resonator group
Ratio of top to bottom 20%	Pitches of inner 2 satellite resonators
Ratio of top to bottom 10%	Pitches of outer 2 satellite resonators (missing values here become clusters)
Water access	Resonator decay times (short is dry)
Distance from ship	Volume, attack (far away is blurred)
Direction relative to ship	Spatial direction in speaker ring
Ship speed, direction, winds	Direction, timbre, and volume of windlike noise

Figure 13.16 shows the sound design: the 10 percent and 20 percent ratios are converted to satellite frequencies symmetrically above and below the root; the exciter signal is composed of individual random triggers for each resonator, some common pulses, and a noise halo to help perceptual fusion of this aggregate. The resonators use Formlet UGens so their attack can be softened, which is used to render sound source distance.

```
(
Ndef(\single, {|rootFreq = 220, outProp = 4.5, inProp = 2.2,   atta
ck=0.00, decay = 1.0, dens = 2, amp=0.2, step=0.33333|

   var freqs = rootFreq * [1/outProp, 1/inProp, 1, inProp,   outPro
p];

   // five individual rd triggers for
```

```
    // each component, weighted for center
var exciter = (Dust2.ar(dens * [1,2,4,2,1] * 0.07)
        + Dust2.ar(dens * 0.3))     // + some common attacks
              .clip2(0.5)
              .lag(0.0003)        // slightly filtered
              * (dens ** −0.5)   // amplitude comp for dust density
        + PinkNoise.ar(0.002)    // some fused background noise
        * (decay ** −0.5);       // amplitude comp. for decay
    var resonator = Formlet.ar (exciter, freqs,
        Ramp.kr(attack, step), decay,
        AmpComp.kr(freqs.max(50))
    ).softclip.sum;

    LeakDC.ar(resonator, 0.95);
}).play;
)
```

Figure 13.16
Sound design for a single country.

The code example PlaySingleCountries.scd (in the book code repository) allows direct comparison between countries by switching to the parameters one country at a time. The full piece itself as code and a headphone-rendered audio file are also provided.

13.7.3 "Terra Nullius"

Social data are measured in many different ways; their numerous dimensions (such as population density, life expectancy, illiteracy) may reveal interesting correlations and give insight into human life on this planet. However, not every factor is measured equally everywhere. When experimenting with the ICAD 2006 world social data at the Science by Ear workshop,[39] we wrote an event-based system for experiments with combinations of data dimensions. To give the events temporal order, we sorted them by one given dimension and along this axis sonified the magnitudes of a number of other dimensions (for instance, sorted by GDP and then interpreted access to drinking water as a frequency value).

But what should we do with data sets that lack the measurement that we want to sort them by? How do we sonify absence? We can leave out that data point, but this affects the comparability of the rest of the parameters; we may interpolate missing data, but from which neighbors? Just as the voice of the excluded cannot be counted, there is no general law for correctly substituting missing data. The piece "Terra Nullius" takes this fundamental problem as a starting point (Rohrhuber, 2006).

Imagine a set of countries within a zone parallel to the equator, starting with latitudes near England (see figure 13.17). Instead of sonifying existing data values, only the missing dimensions of each country are sonified: A broadband noise source is filtered into 48 bands, each representing a single data dimension. When the country currently being sonified has a missing value in a dimension, the corresponding noise band becomes audible.

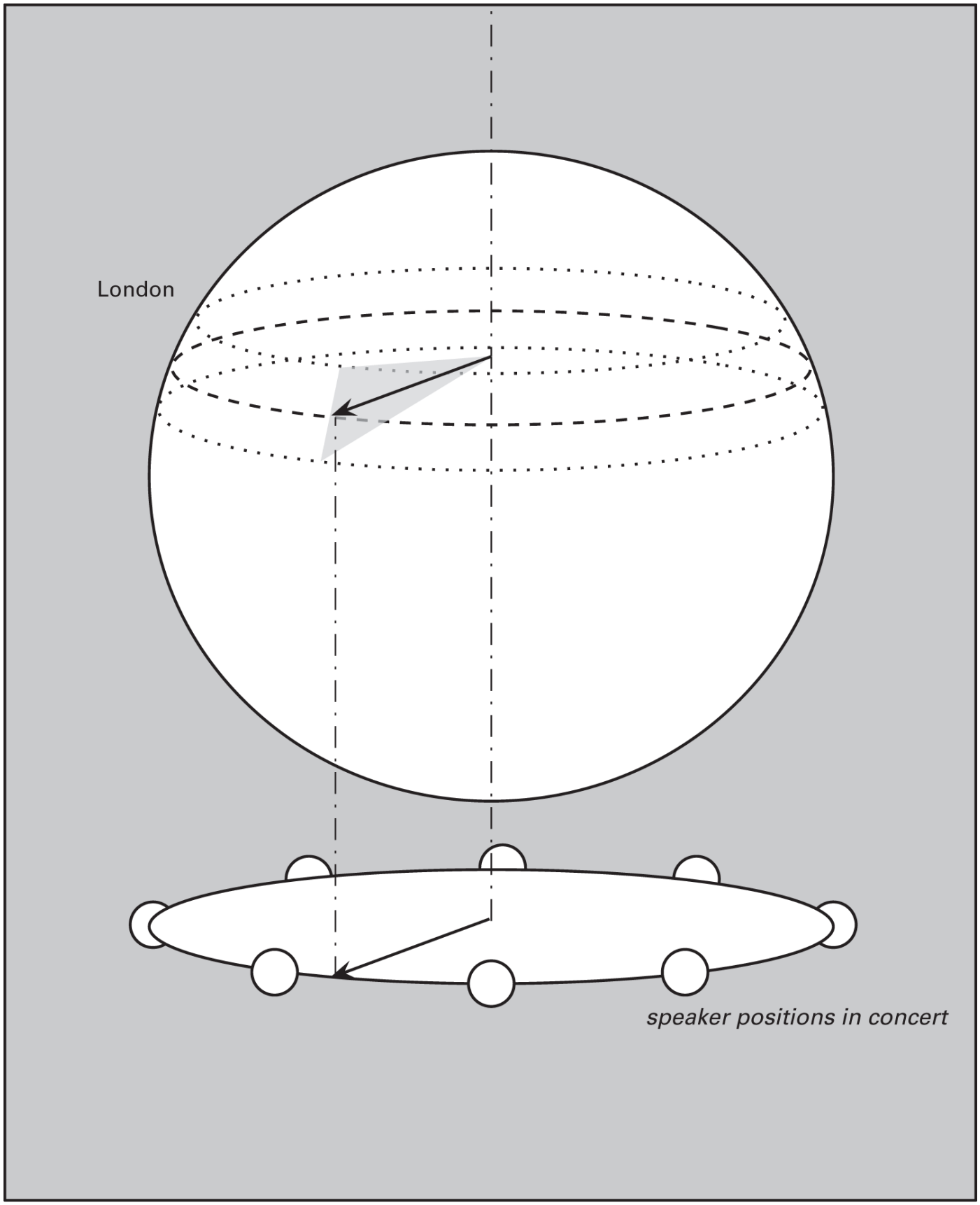


Figure 13.17
Terra Nullius: moving latitude zone on the globe.

We spin the imaginary globe and move eastward: all countries inside the zone are sonified one at a time, in order of their longitude. The noise spectrum pans according to longitude around the ring of speakers. With every round, the latitude slice becomes wider and wider until all countries are included in each round.

Then the zone narrows again, this time sonified with smaller filter bandwidths, so the dimensions progressively separate more. We end with the latitude and longitude of London. Over time, a background noise on all speakers has slowly faded and remains as the ending.

Technically, a `NodeProxy` (see chapter 7) plays the noise bands, and the amplitudes of each spectral band, the pan position, and the background noise level are set from a task that iterates over the data. Some sorting and task functions are defined: `q.makeSelection(10, 40)` selects countries by latitude band (here, within 10 degrees of 40 degrees north); `q.makeOrder(9)` sorts by population density; and `q.makeCycle(0.04).play` creates a routine with a delta time of 0.04 and plays once around the globe. In the code file (included in the website materials), these functions can be used to experiment with the sonification. `Tdef(\x).play` plays the entire piece.

Finally, "Terra Nullius" can be regarded as a meditation on the status of missing points in a generalized geometry, as Bernhard Riemann proposed in 1854. Riemann suggested abandoning the conventional idea of a point, and by relying only on quantitative and qualitative *Bestimmungsweisen* (translated as "modes of determination"), he redefined the concept of dimensionality. The former (e.g., different degrees of temperature) can be mutually ordered, but the latter (such as temperature with speed) can't. Now missing data are *quantitative* in the sense that it is a possible measurement; if we forget to measure temperature, we still forget to measure *temperature*; by consequence, it should be sortable. On the other hand, since it is missing, it has no measure and nothing to specify an order, just like any *qualitative* dimension. There is no solution to this dilemma. We can only stop here and later come back to the only available complete data—geographical location, a consequence of the very act of measuring. To characterize this *terra nullius* by different shades of filtered noise is perhaps the only compromise between quality and quantity: the missing dimension itself is represented by its position in the spectrum, and the absence of the data is represented by the presence of noise; we are left with only the sonification of the act of measuring and its incompleteness.

13.8 Conclusion

Shifts in scientific worldview have always been accompanied by changes in methods and representations, just as inventions of representation have caused paradigm shifts, such as Riemann's generalization of geometry, which became a foundation for Einstein's

space-time. In a sense, sonification opens a different perspective on things: namely, listening. For the sciences, this may imply reconsidering underlying premises and rethinking just what one is looking (and listening) for; it also may lead to novel ways of didactic representation. For the sonic arts, sonification brings about curiosity about both the origins and potential meanings of sounds and broader cultural and social contexts. Perhaps sonification is to sound art what documentary is to film. Certainly, it is an area open for exploration, full of inspiration and potential insights for sciences and the arts. SuperCollider has become one of the main platforms for sonification research, providing a system in which transdisciplinary cooperations that are joyful and rigorous in equal measure are possible—and necessary.

Notes

1. See http://www.icad.org.
2. https://gvu.gatech.edu/research/labs/sonification-lab.
3. https://smartech.gatech.edu/handle/1853/49750.
4. https://www.cit-ec.de/en/ami/.
5. http://sonenvir.at.
6. https://qcd-audio.at/, https://sysson.iem.at/.
7. https://projectradical.github.io/about/.
8. https://sonification.de/handbook/.
9. https://sonification.design/.
10. https://www.icad.org/websiteV2.0/Conferences/ICAD2004/concert.htm.
11. https://sonification.net/site/.
12. https://ctm-festival.de.
13. https://xcoax.org/.
14. For example, https://audible-data-streams.com/, a sonification festival held in Berlin in 2017; and https://www.thesoundofdata.lu/about-s-o-d/, an art/science festival held in Luxemburg in 2022.
15. https://www.ligo.caltech.edu/video/ligo20171016v5.
16. https://www.system-sounds.com/.
17. See http://www.sonification.de and http://www.techfak.uni-bielefeld.de/ags/ni.
18. See http://sonenvir.at.
19. https://sysson.kug.ac.at/index.php?id=14007.
20. https://sonenvir.at/workshop/ and https://qcd-audio.at/sbe2.html.
21. https://sonification.de/handbook/chapters/chapter10/.
22. Over the years, a number of Python-integrations have been developed. Early projects are SoniPy von D. Worrall (https://link.springer.com/chapter/10.1007/978-3-030-01497-1_6). More recently, the sc3nb framework (Hermann et al., 2021) allows for exchanging data between Python and SC and making the SC audio server directly accessible with Python via the respective Python classes (https://interactive-sonification.github.io/sc3nb/stable/). Recently, a port of the Super Collider's class library to Python also has been developed. (Samaruga et al., 2022). Another promising project is Josiah Wolf Oberholtzer's Supriya application programming interface (API; https://pypi.org/project/supriya/).
23. Listening examples for seismological data can be found at https://phaidra.kug.ac.at/search#?page=1&pagesize=10&collection=o:92490.
24. https://www.pflanzenforschung.de/de/pflanzenwissen/plantisonics/plantainment-11#.
25. The data are taken from the *exest.csv* file, downloaded from http://www.ojp.usdoj.gov/ bjs/cp.htm on December 29, 2007. Current data can be obtained at https://bjs.ojp.gov/library/publications/prisoners-executed.
26. This was written for the networked ensemble powerbooks_unplugged; see https://web.archive.org/web/20120216040047/http://pbup.goto10.org/.

27. This is included in the SuperCollider plug-ins: https://github.com/supercollider/sc3-plugins
28. https://christinakubisch.de/.
29. https://www.finetuned.org/valentina-vuksic.html.
30. https://www1.wdr.de/radio/wdr3/programm/sendungen/wdr3-studio-akustische-kunst/wandering-lake-an-atomic-opera-von-echo-ho-und-ulrike-janssen-100.html.
31. The same data were used at the second Science by Ear Workshop, and different approaches by interdisciplinary research teams can be found here: https://qcd-audio.at/sbe2/sbe_tpc.html.
32. Lily Asquith, "Listening to Data from the Large Hadron Collider | Lily Asquith |TEDxZurich," YouTube, December 2013, youtu.be/iQiPytKHEwY
33. https://carlascaletti.com/sounds/sound-art/quantum/ and https://www.gillesjobin.com/en/creation/quantum/.
34. http://piethopraxis.org/projects/bonner-durchmusterung/index.html.
35. https://en.wikipedia.org/wiki/Durchmusterung.
36. See https://github.com/supercollider-quarks/quarks.
37. See the ICAD 2006 concert call at http://www.dcs.qmul.ac.uk/research/imc/icad2006/concert.php, and the website for the papers and audio files of the pieces.
38. Both the art of navigation outside Europe, especially in Polynesia, and the history of knowledge acquisition about navigation, maps, and mapmaking are covered by Conner (2005).
39. See http://sonenvir.at/workshop for documentation of this.

References

Andreopoulou, A., and V. Goudarzi. 2021. "Sonification First: The Role of ICAD in the Advancement of Sonification-Related Research." In *ICAD Proceedings*.

ASA demos by Al Bregman: http://webpages.mcgill.ca/staff/Group2/abregm1/web/downloadsdl.htm.

Baier, G., T. Hermann, and U. Stephani. 2007. "Event-Based Sonification of EEG Rhythms in Real Time." *Clinical Neurophysiology*, 118(6): 1377–1386.

Ballora, M. 2000. "Data Analysis through Auditory Display: Applications in Heart Rate Variability." PhD thesis, McGill University, Montreal.

Barrass, S. 1997. "Auditory Information Design." PhD thesis, Australian National University.

Beilharz, K., and S. Ferguson. 2009. "An Interface and Framework Design for Interactive Aesthetic Sonification." In *ICAD Proceedings*.

Bijsterveld, K., and S. Krebs. 2013. "Listening to the Sounding Objects of the Past: The Case of the Car." *Sonic Interaction Design*, 3–38. https://orbilu.uni.lu/handle/10993/21143.

Bovermann, T., J. Groten, A. de Campo, and G. Eckel. 2007. "Juggling Sounds." In *Proceedings of the 2nd International Workshop on Interactive Sonification*.

Bovermann, T., J. Rohrhuber, and A. de Campo. 2011. "Laboratory Methods for Experimental Sonification." In T. Hermann, A. Hunt, and J. G. Neuhoff, eds., *The Sonification Handbook* (pp. 237–272). Logos Verlag Berlin GmbH.

Bregman, A. S. 1990. *Auditory Scene Analysis: The Perceptual Organization of Sound*. Cambridge, MA: MIT Press.

Cherston, J., E. Hill, S. Goldfarb, and J. A. Paradiso. (2016, May). "Sonification Platform for Interaction with Real-Time Particle Collision Data from the Atlas Detector." In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (pp. 1647–1653).

Conner, C. D. 2005. *A People's History of Science: Miners, Midwives and "Low Mechanicks."* Nation Books.

de Campo, A. 2009. "Science by Ear: An Interdisciplinary Approach to Sonifying Scientific Data." PhD thesis, University for Music and Dramatic Arts, Graz, Austria.

de Campo, A. 2007. "Toward a Data Sonification Design Space Map." In *ICAD Proceedings*.

de Campo, A., and C. Dayé. 2006. "Navegar é preciso, viver não É preciso." In *ICAD Proceedings*.

de Campo, A., and M. Egger de Campo. 1999. "Sonification of Social Data." In *Proceedings of ICMC 1999*.

de Campo, A., C. Frauenberger, K. Vogt, A. Wallisch, and C. Dayé. 2006. “Sonification as an Interdisciplinary Working Process.” In *ICAD Proceedings*.

de Campo, A., A. Wallisch, R. Hoeldrich, and G. Eckel. 2007. “New Sonification Tools for EEG Data Screening and Monitoring.” In *ICAD Proceedings*.

De Mol, L. 2007. “Tracing Unsolvability: A Mathematical, Historical and Philosophical Analysis with a Special Focus on Tag Systems.” PhD thesis, University of Ghent, Belgium.

Dombois, F. 2001. “Using Audification in Planetary Seismology.” In *Proceedings of the 7th International Conference on Auditory Display* (ICAD).

Dombois, F. 2008. “Sonifyer a Concept, a Software, a Platform.” In *ICAD Proceedings*.

Drake, S. 1975. “The Role of Music in Galileo’s Experiments.” *Scientific American,* 232(5): 98–104.

Enge, K., A. Rind, M. Iber, R. Höldrich, and W. Aigner. 2021. “It’s About Time: Adopting Theoretical Constructs from Visualization for Sonification.” In *Audio Mostly,* (pp. 64–71).

Enge, K., A. Rind, M. Iber, R. Höldrich, and W. Aigner. 2023. “Towards a Unified Terminology for Sonification and Visualization.” *Personal and Ubiquitous Computing*, 27: 1949–1963.

Fitch, W. T., and G. Kramer. 1994. “Sonifying the Body Electric: Superiority of an Auditory Over a Visual Display in a Complex Multivariate System.” In G. Kramer, ed., *Auditory Display* (pp. 307–326). Reading, MA: Addison-Wesley.

Fletcher, H., and W. A. Munson. 1933. “Loudness, Its Definition, Measurement, and Calculation.” *Journal of the Acoustical Society of America,* 5: 82–108.

Flowers, J. H. 2005. Thirteen Years of Reflection on Auditory Graphing: Promises, Pitfalls, and Potential New Directions. In *ICAD Proceedings*.

Frauenberger, C., and T. Stockman. 2006. “Patterns in Auditory Menu Design.” In *ICAD Proceedings*.

Gaver, W. W., R. B. Smith, and T. O’Shea. 1991. “Effective Sounds in Complex Systems: The ARKola Simulation.” In *Proceedings of CHI ’91* (pp. 85–90). New York: ACM Press.

Giordano, B. L., P. Susini, and R. Bresin. 2013. “Perceptual Evaluation of Sound-Producing Objects.” In K. Franinovic and S. Serafin, eds., *Sonic Interaction Design* (pp. 151–197). Cambridge, MA: MIT Press.

Goudarzi, V., H. H. Rutz, and K. Vogt. 2013. “User Centered Audio Interface for Climate Science.” In *Proceedings of Interactive Sonification Workshop,* Erlangen, Germany.

Groß-Vogt, K., K Enge, and I m. zmölnig. 2023. “Reflecting on qualitative and quantitative data to frame criteria for effective sonification design.” In *Proceedings of the 18th International Audio Mostly Conference 2023*. New York, NY, USA, 2023. Association for Computing Machinery. http://dx.doi.org/10.1145/3616195.3616233

Groß-Vogt, K., M. Frank, and R. Höldrich. 2020. “Focused Audification and the Optimization of Its Parameters.” *Journal on Multimodal User Interfaces*, 14: 187–198.

Hermann, T. 2008. “Taxonomy and Definitions for Sonification and Auditory Display.” *International Community for Auditory Display*.

Hermann, T., A. Hunt, and J. G. Neuhoff. 2011. *The Sonification Handbook* (Vol. 1). Berlin: Logos Verlag.

Hermann, T., and D. Reinsch. (2021, September). “sc3nb: A Python-SuperCollider Interface for Auditory Data Science.” In *Proceedings of the 16th International Audio Mostly Conference* (pp. 208–215).

Hayward, C. 1994. “Listening to the Earth Sing.” In G. Kramer, ed., *Auditory Display* (pp. 369–404). Reading, MA: Addison-Wesley.

Hermann, T. 2002. “Sonification for Exploratory Data Analysis.” PhD thesis, Bielefeld University, Germany.

Hermann, T., G. Baier, U. Stephani, and H. Ritter. 2006. “Vocal Sonification of Pathologic EEG Features.” In *ICAD Proceedings*.

Hinterberger, T., and G. Baier. 2005. “POSER: Parametric Orchestral Sonification of EEG in Real-Time for the Self-Regulation of Brain States.” *IEEE Multimedia,* special issue on sonification, 12(2): 70–79.

Höldrich, R., and K. Vogt. 2010. “A Metaphoric Sonification Method—Towards the Acoustic Standard Model of Particle Physics.” In *ICAD Proceedings*.

Kowalski, H., L. Lipatov, D. Ross, and G. Watt. 2010. “Using HERA Data to Determine the Infrared Behaviour of the BFKL Amplitude.” *European Physical Journal C—Particles and Fields*, 1–16. https://doi.org/10.1140/epjc

/s10052-010-1500-6.

Kramer, G., ed. 1994a. *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Reading, MA: Addison-Wesley.

Kramer, G. 1994b. "An Introduction to Auditory Display." In G. Kramer, ed., *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Reading, MA: Addison-Wesley.

Lutters, B. T., and P. J. Koehler. 2017. "Brainwaves in Concert: The 20th Century Sonification of the Electroencephalogram.: *Neurology*, 4(88).

Mardakheh, M., and W. Scott. 2022. "A Strata-Based Approach to Discussing Artistic Data Sonification." *Leonardo*, 55 (5): 516–520. https://doi.org/10.1162/leon_a_02257.

McKusick, V. A., W. D. Sharpe, and A. O. Warner. 1957. "Harvey Tercentenary: An Exhibition on the History of Cardiovascular Sound Including the Evolution of the Stethoscope." *Bulletin of the History of Medicine*, 31: 463–487.

Nees, M. A. 2018. "Auditory Graphs are Not the "Killer App" of Sonification, But They Work." *Ergonomics in Design*, 26(4): 25–28.

Nees, M. A. 2019. "Eight Components of a Design Theory of Sonification." In *ICAD Proceedings*.

Neuhoff, J. G., ed. 2004. *Ecological Psychoacoustics*. Elsevier Academic Press.

Neuhoff, J. G. 2019. *Is Sonification Doomed to Fail?* Georgia Institute of Technology.

Parseihian, G., C. Gondre, M. Aramaki, S. Ystad, and R. Kronland-Martinet. 2016. "Comparison and Evaluation of Sonification Strategies for Guidance Tasks." *IEEE Transactions on Multimedia*, 18(4): 674–686.

Pauletto, S., and A. Hunt 2004. "A Toolkit for Interactive Sonification." In *ICAD Proceedings*.

Pereverzev, S. V., A. Loshak, S. Backhaus, J. C. Davies, and R. E. Packard. 1997. "Quantum Oscillations Between Two Weakly Coupled Reservoirs of Superfluid 3 He." *Nature*, 388(449): 449–451.

Pigafetta, A. 1530 (reprint 1800). *Primo viaggio intorno al globo terracqueo.* Milan: Giuseppe Galeazzi.

Pigafetta, A. 2001 (orig. 1525). *Mit Magellan um die Erde.* Lenningen, Germany: Edition Erdmann.

Riemann, B. [1854] 1867. "Über die Hypothesen, welche der Geometrie zugrunde liegen." *Abhandlungen der Königlichen Gesellschaft der Wissenschaften zu Göttingen,* 13.

Riess, F., and P. Heering. 2005. "Reconstructing Galileo's Inclined Plane Experiments for Teaching Purposes." In *Proceedings of the 8th Conference on International Philosophy, Sociology and Science Teaching.*

Rohrhuber, J. 2006. "Terra Nullius." In *ICAD Proceedings*.

Rohrhuber, J. 2011. "Operator, Operation. Sehen was das Photon sieht." In M. Bierwirth, O. Leisert, and R. Wieser, eds., *Ungeplante Strukturen: Tausch und Zirkulation*. Munich: Fink Verlag.

Rohrhuber, J. 2018. "Algorithmic Music and the Philosophy of Time." In A. McLean and R. T. Dean, eds., *The Oxford Handbook of Algorithmic Music* (chap. 2, pp. 17–40). Oxford: Oxford University Press.

Rosenboom, D. 1990. *Extended Musical Interface with the Human Nervous System.* San Francisco: Leonardo Monograph Series.

Rutz, H. H., K. Vogt, and R. Höldrich. 2015. "The SysSon Platform: A Computer Music Perspective of Sonification." In *ICAD Proceedings*.

Samaruga, L., and P. Riera. (2022, September). "A Port of the SuperCollider's Class Library to Python." In *ICAD Proceedings*.

Schoon, A., and A. Volmar. 2014. *Das geschulte Ohr. Eine Kulturgeschichte der Sonifikation*. Bielefeld: transcript Verlag.

Serafin, S., K. Franinović, T. Hermann, G. Lemaitre, M. Rinott, and D. Rocchesso. 2011. "Sonic Interaction Design." In *The Sonification Handbook*.

Speeth, S. D. 1961. "Seismometer Sounds." *Journal of the Acoustical Society of America,* 33: 909–916.

Sterne, J., and M. Akiyama. 2011. "The Recording that Never Wanted to Be Heard and Other Stories of Sonification." In *The Oxford Handbook of Sound Studies*. https://doi.org/10.1093/oxfordhb/9780195388947.013.0115
.

Straebel, V., and W. Thoben. 2014. "Alvin Lucier's Music for Solo Performer: Experimental Music beyond Sonification." *Organised Sound*, 19(1): 17–29. https://doi.org/10.1017/S135577181300037X.

Supper, A. 2012. “Lobbying for the Ear: The Public Fascination With and Academic Legitimacy of the Sonification of Scientific Data.” PhD thesis, Datawyse/Universitaire Pers Maastricht. https://doi.org/10.26481/dis.20120606as.

Supper, A. 2014. “Sublime Frequencies: The Construction of Sublime Listening Experiences in the Sonification of Scientific Data.” *Social Studies of Science*, 44(1): 34–58.

Urick, R. J. 1967. *Principles of Underwater Sound*. New York: McGraw-Hill.

Väljamäe, A., T. Steffert, S. Holland, et al. 2013. “A Review of Real-Time EEG Sonification Research.” In *ICAD Proceedings*.

Vasilakos, K., S. Wilson, T. McCauley, T. Yeung, E. Margetson, and M. Khosravi. 2020. “Sonification of High Energy Physics Data Using Live Coding and Web Based Interfaces.” In *NIME Proceedings* [online], pp. 464–470.

Vickers, P., B. Hogg, and D. Worrall. 2017. “Aesthetics of Sonification: Taking the Subject-Position.” In Body, *Sound and Space in Music and Beyond: Multimodal Explorations* (pp. 89–109). Routledge.

Vogt, K. 2010. “Sonification of Simulations in Computational Physics.” PhD thesis, KUG Graz, Austria.

Vogt, K., D. Pirrò, I. Kobenz, R. Höldrich, and G. Eckel. 2010a. “PhysioSonic-Evaluated Movement Sonification as Auditory Feedback in Physiotherapy.” In *Auditory Display: 6th International Symposium*, CMMR/ICAD 2009, Copenhagen, May 18–22, 2009 (pp. 103–120). Berlin: Springer Berlin Heidelberg.

Vogt, K., R. Höldrich, D. Pirro, M. Rumori, S. Rossegger, W. Riegler, and M. Tadel. 2010b. “A Sonic Time Projection Chamger: Sonified Particle Detection at CERN.” In *ICAD Proceedings*.

Vogt, K., V. Goudarzi, and R. Parncutt. 2013. “Empirical Aesthetic Evaluation of Sonifications.” In *ICAD Proceedings*.

Volmar, A. 2013. “Listening to the Cold War: The Nuclear Test Ban Negotiations, Seismology, and Psychoacoustics, 1958–1963.” *Osiris: Music, Sound, and the Laboratory from 1750–1980*, 28(1): 80–102.

Walker, B. N. 2002. “Magnitude Estimation of Conceptual Data Dimensions for Use in Sonification.” *Journal of Experimental Psychology: Applied*, 8(4): 211.

Walker, B. N., and J. T. Cothran. 2003. *Sonification Sandbox: A Graphical Toolkit for Auditory Graphs*. Atlanta: Georgia Institute of Technology.

Wedensky, N. 1883. “Die telefonische Wirkungen des erregten Nerven” (The Telephonic Effects of the Excited Nerve). *Centralblatt für medizinische Wissenschaften*, XXI(26): 465–468.

Worrall, D. 2018. “Sonification: A Prehistory.” In *ICAD Proceedings*.

Worrall, D. 2019. *Sonification Design. From Data to Intelligible Soundfields*. Human–Computer Interaction Series. Springer.

Zotter, F., M. Zaunschirm, M. Frank, and M. Kronlachner. 2017. “A Beamformer to Play with Wall Reflections: The Icosahedral Loudspeaker.” *Computer Music Journal*, 41(3): 50–68. https://doi.org/10.1162/comj_a_00429.

Zweig, S. 1938. *Magellan—Der Mann und seine Tat.* Frankfurt am Main, 1983.

14 Spatialization with SuperCollider

Marije A. J. Baalman and Scott Wilson

14.1 Spatial Sound

The notion of spatial sound is a tautology, as sound is a spatial phenomenon in itself. Sound is a wave phenomenon, which means that it is not in one specific location but rather fills the medium, although its strength at certain points may vary. Sound can be caused by the movement of an object and can be transported in a solid, gas, or fluid.

If there is a sound source at a certain position in space, the acoustic wave will propagate outward in all directions. Only when there are changes in the medium (such as an obstacle, wind, or, more generally, a transition to another medium) will the propagation direction change. At the borders between media, part of the energy of the wave will be transmitted into the other medium, and part of it will be reflected. An example of transmitted energy would be a discotheque whose windows are vibrating in response to loud music. What you hear coming through the windows has been transmitted. Reflections create what is usually referred to as "room acoustics."

Within room acoustics, reflections are distinguished in three zones within the impulse response: very early reflections (also referred to as "pseudodirect sound"), up to about 5 ms after the direct sound; early reflections, up to 80–100 ms; and reverb, everything above 80–100 ms. For the very early and early reflections, we can distinguish, or at least have a sense of, the direction from which the reflection is coming; this is the information which helps us get an idea of the location of the sound (especially the distance) and also the size of the sounding object. The reverb gives us an overall sense of the acoustics of the room and a feeling of envelopment. The ratio between the level of the direct sound and the reverb gives us a sense of the presence of the sounding object. For reverb, we can distinguish around 8 directions (Sonke and De Vries, 1997).

14.2 Spatial Perception

Our spatial perception has its biological roots as a warning system: we need to know where a sound comes from so we can direct our visual attention to it and determine whether the sound is coming from something that we can eat or from something that wants to eat us (or to warn us of any other danger). We can distinguish spatial

properties of sound because we have two ears, so we can compare the two different signals. Generally, we can compare the time difference, called the "interaural time difference (ITD)," and the level difference, the "interaural level difference (ILD)," of the signals that reach our ears; the first is a result of the difference in the path length that a sound has traveled to each ear, and the second is a result of both the difference in path length and the masking effect of the head. Our spatial perception can be quite accurate in the horizontal plane, down to 1 or 2 degrees right in front of the head and varying somewhat, depending on the type of sound. In the vertical plane, the accuracy is a bit more crude (about 3 to 5 degrees). It is generally understood that the perception in the horizontal plane is based primarily on the ITD and the ILD, whereas in the vertical plane, the shape of our ears plays a role, as it filters the sound in a specific way (Blauert, 1997). Furthermore, head movements help us to locate a sound.

Apparent source width is often related to the "interaural cross-correlation coefficient (IACC)" (Ando, 1985), which is a measure of how closely the two ear signals are correlated with one another. Reflections help us determine the location of a sound; they also give us a feeling of the presence of and envelopment by the sound. Finally, different locations of sounds help us distinguish between different streams of sound. This is one aspect of what is generally known as the "cocktail party effect." That is, even when everyone around us is talking, we can focus our attention on our conversation partner and choose to listen only to what he or she says. This effect can be used similarly in composition: different melodies played from different locations will be perceived as separate melodies, whereas if they come from the same location, they may be perceived as a single, more complex melody. Put another way, spatial location helps us to segregate audio into separate streams of information.

14.3 Spatial Composition Techniques and Spatial Audio Technologies

Based on theories of acoustic sound propagation and spatial sound perception, researchers have developed different technologies to create realistic spatial sound images. A reproduction enabled by this might be an acoustic concert that is accurately recorded and played back in another space.

As an electronic music composer, you may want to exploit the advantages and limitations of different spatial audio technologies, to create a natural-like acoustic event, or to confuse the listener's normal spatial assumptions (Baalman, 2010).

14.4 Standard Spatial Techniques in SuperCollider

Before creating multichannel sound with SuperCollider, you must configure the server to output an appropriate number of channels to your sound card:

```
// recall s refers to the default server; if you have 16 channels t
o output:
s.options.numOutputBusChannels = 16;
s.boot; // must configure this before booting
```

14.4.1 Multichannel Signals in SuperCollider

Multichannel signals in SuperCollider are represented as Arrays. Although a detailed discussion of Arrays in SC (and, indeed, of its extensive Collections support in general) is beyond the scope of this chapter, the reader is encouraged to become familiar with this aspect of SC, as it is very useful for the manipulation (i.e., mixing, routing, etc.) of multiple channels. Multichannel expansion (see chapter 1 and the Multichannel Expansion Help file) provides an elegant way to create multichannel signals.

14.4.2 Monophony

The simplest way to place a sound somewhere spatially is to let it come from a specific loudspeaker. When you have several speakers, this is called "multiple mono." It is a method which will always result in a perceptually correct spatialization, as the loudspeaker functions as a point source. Multiple mono can be used for serial compositional techniques in which the location of the sound is serialized. Another use can be stream segregation as described in chapter 13 (also see figure 14.1).

```
(
/// a SynthDef to use
SynthDef (\nicepoc, {|out=0,freq=440,amp=0.1,dur=0.3|
   Out.ar (out, SinOsc.ar (freq, mul: amp) * EnvGen.kr   (Env.perc
(0.05,1), timeScale: dur, doneAction:2))
   }).add;
)

// mono, 1 channel:
(
p = Pbind(
   \degree, Pseq([0, 3, 5, 6, 7],5),
   \dur, 0.2,
   \instrument, \nicepoc
).play;
)

// multiple mono:
// the melody gets played on both channels, the second note in the
 pattern differs,
```

```
// so when listening to it, the space "spreads" out
(
p = Pbind(
    \degree, Pseq([0, [3,4], 5, 6, 7],5),
    \out, [0,1],
    \dur, 0.2,
    \instrument, \nicepoc
).play;
)
```

Figure 14.1
Mono and multiple mono.

14.4.3 Pairwise Panning (Stereophony) and Its Extensions

Stereophony is a way to simulate a spatial field by sending two different signals to two speakers. The optimal listening position should subtend an angle of 60 degrees between the speakers, or +/−30 degrees from the median plane, as shown in figure 14.2. It is possible to simulate the positioning of a sound between these two speakers through *panning* techniques. Many popular surround sound techniques have been derived from stereophony, and they extend the pairwise panning paradigm to larger arrays of loudspeakers. Examples include quadrophony (usually where one speaker is placed in each corner of a square, but sometimes in other configurations); octophony (most often adjacent speakers at 45-degree angles from each other); and cinema formats such as 5.1 (a normal stereo pair with an additional center speaker for dialogue, a subwoofer which plays an optional "low-frequency effects" channel, and two rear surround channels, usually at approximately +/−110 degrees from the median plane), 7.1 (like 5.1 but with two additional speakers at approximately +/− 90 degrees), and Dolby Atmos (an object-oriented approach enabling adaptation to various setups, but usually with overhead speakers in loudspeaker setups).[1]

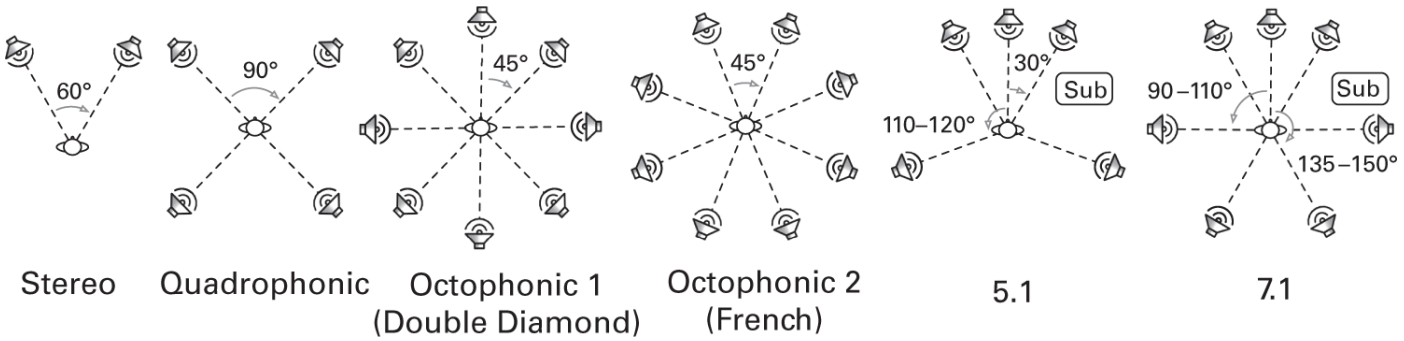


Figure 14.2
Common multichannel configurations, from left to right: stereo, quadrophonic, octophonic, and 5.1.

The normal technique for panning a signal between two speakers is as follows. If a sound is to appear to come from between two speakers, each speaker gets a proportion in the level of that signal, thus simulating an appropriate ILD. This is normally done with equal power between the two channels rather than equal amplitude (i.e., a linear cross-fade) between them. The former is perceptually smooth, whereas the latter has an audible dip of −3 dB in amplitude when a mono signal is panned to the center due to in-room phase cancellation.

Although effective under controlled circumstances, pairwise panning has some important limitations. (See Rumsey, 2001, for a detailed discussion of this topic and of spatial audio in general.) One is of particular importance: the Precedence Effect, or the Law of the First Wave Front. Pairwise panning assumes that the listener is equidistant from each of the two loudspeakers. If the ITD that results from not being equidistant is greater than about 1 ms, the panning illusion will collapse and the listener will tend to locate the sound in the closer speaker. This has strong implications for multichannel setups, particularly in concert situations where the position of some listeners may be less than ideal. In general, the closer the two speakers are to one another, and the narrower the angle between them relative to the listener's position, the less likely this is to be a problem. Similarly, many multichannel implementations attempt to use pairwise panning between distant and closer speakers, or to pan through the middle of a ring of speakers (i.e., the audience area). Although this may achieve a general effect of movement, attempts to simulate locations between the speakers in such cases are unlikely to be thoroughly convincing.

In SuperCollider, there are a number of pairwise panning unit generators (UGens). Figure 14.3 shows these UGens and their arguments. The two basic channel panners are `Pan2` and `LinPan2` (the "reverse" versions of these are `XFade2` and `LinXFade2`, to mix two inputs to one channel). With the UGen `Balance2`, you can correct the balance of a stereo signal. `Rotate2` rotates a sound field by taking two inputs and rotating them with angle pos; this UGen is actually meant for Ambisonics (see below), but it also gives interesting effects on a stereo signal.

```
// 2 channel panners:
Pan2.ar (in, pos, level);
LinPan2.ar (in, pos, level);
Balance2.ar (left, right, pos, level);
Rotate2.ar (x, y, pos);

// 4 channel panner:
Pan4.ar (in, xpos, ypos, level);

// N-channel panner:
```

```
PanAz.ar (numChans, in, pos, level, width, orientation);

// spread M channels over a stereo field:
Splay.ar (inArray, spread, level, center, levelComp);

// spread M channels over N channels:
SplayAz.ar (numChans, inArray, spread, level, width, center,   orie
ntation, levelComp);
```

Figure 14.3
Overview of the different panners in SuperCollider with their argument names.

`Pan4` is a quadraphonic panner which pans between left-right and front-back. Note that for reasons to do with the precedence effect, it cannot effectively simulate locations *within* the speaker array.

`PanAz` is a multichannel panner with an arbitrary number of channels. It is assumed that the speakers are laid out in a circle at equally spaced angles. The first argument is the number of channels, the second argument is the input signal, and the third is the position in the circle. Channels are evenly spaced over a cyclic period of 2.0, with 0.0 equal to the position directly in front; `2.0/numChans` is a clockwise shift `1/numChans` of the way around the ring (and `−2.0/numChans` is a counterclockwise shift by one speaker); `4.0/numChans` is equal to a shift of `2/numChans`; and so on. The `width` (the fifth argument) is the width of the panning envelope. Normally, this is 2.0, which pans between adjacent pairs of speakers. If it is greater than 2.0, the pan will be spread over more speakers, and if it is less than 1.0, it will leave gaps between the speakers. Note that widths greater than 2.0 will not effectively simulate expanded source width as the signals are correlated (see discussion of IACC above), but they will increase localization blur, which may be useful in some circumstances. The orientation should be 0 when the first speaker is directly in front and 0.5 when the front is halfway between two speakers; thus the position directly in front may or may not correspond to a speaker, depending on the setting of this argument. (See also figure 14.4.)

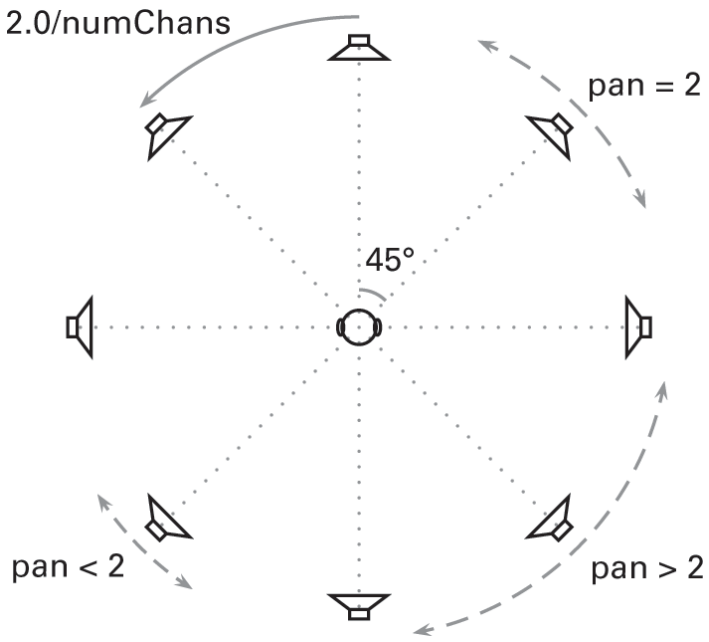


Figure 14.4
Visualization of the arguments of `PanAz`.

The `PanX` UGen (available as part of the sc3-plug-ins; see http://supercollider.github.io/sc3-plugins/) allows you to pan over a linear array of speakers or, if you nest them, over a grid of speakers. This can be useful if you are working with a speaker setup that is not surrounding an audience, but for example want to create sounds that move along a passageway, or that move across a grid of speakers mounted in a ceiling, floor, or vibrating bed.

`Splay` spreads an Array of channels across a stereo field. The first argument is an `Array` with input channels, and the second argument determines the spread of the channels, where 0 is no spread and 1 is a complete spread. The center argument determines the central point around which the channels are spread. `SplayAz` does the same thing for a circular setup of equally spaced loudspeakers, like that assumed in the case of `PanAz`. `SelectX` and `SelectXFocus` can be seen as the reverse versions of these panners, selecting a channel from an array of channels, with equal power cross-fading adjacent channels for between indices.

Some examples using pairwise panner UGens are provided in the website materials.

14.4.4 Delay

With delay lines, you can create single echoes of sound or more complex patterns of early reflections. There are a number of delays available in SuperCollider (see also figure 14.5): `Delay`, `Allpass`, `Comb`, `Tap`, and `PingPong`. The first three are also available in versions that use a `Buffer` for their internal memory: `BufDelay`, `BufAllpass`, and `BufComb`. Most delays are available with a choice of interpolation scheme: cubic (postfix: C, so `DelayC`, etc.), linear (postfix: L), and no interpolation

(postfix: N). Interpolation makes a smoother sound when changing the delay time, but it is computationally more expensive.

```
// single tap delay lines
DelayN.ar(in, maxdelaytime, delaytime, mul, add)
DelayL.ar(in, maxdelaytime, delaytime, mul, add)
DelayC.ar(in, maxdelaytime, delaytime, mul, add)

// allpass filters:
AllpassN.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
AllpassL.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
AllpassC.ar(in, maxdelaytime, delaytime, decaytime, mul, add)

// comb filters (delaylines with feedback):
CombN.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
CombL.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
CombC.ar(in, maxdelaytime, delaytime, decaytime, mul, add)

// buffer versions:

BufDelayN.ar(buf, in, delaytime, mul, add)
BufDelayL.ar(buf, in, delaytime, mul, add)
BufDelayC.ar(buf, in, delaytime, mul, add)

BufAllpassN.ar(buf, in, delaytime, decaytime, mul, add)
BufAllpassL.ar(buf, in, delaytime, decaytime, mul, add)
BufAllpassC.ar(buf, in, delaytime, decaytime, mul, add)

BufCombN.ar(buf, in, delaytime, decaytime, mul, add)
BufCombL.ar(buf, in, delaytime, decaytime, mul, add)
BufCombC.ar(buf, in, delaytime, decaytime, mul, add)

// special delay lines utilising PlayBuf:
Tap.ar(bufnum, numChannels, delaytime)
PingPong.ar(bufnum, inputArray, delayTime, feedback, rotate)
```

Figure 14.5
Delay lines and their arguments.

In the nonbuffer versions, the argument `maxdelaytime` is the maximum delay time that can be used. It must be set when the UGen is created and cannot be changed afterward. In the buffer versions, the size of the buffer determines the maximum delay time. When you want to use long delay times, it is better to use the buffer versions since the allocation of a large buffer in real time (i.e., upon instantiation of a `Synth` using a

delay) is a bad idea; you can also run out of real-time allocatable memory. (You can set the amount that *scsynth* may use with `s.options.memSize` before booting.)

An Allpass filter has no effect on a steady-state sound; you will hear the effect only when you add it to the unfiltered version. A Comb filter, on the other hand, can be used as a resonator. Both can be used to create echoes from an input sound. Examples of these can be found in the Help files.

`Tap` is a delay line implemented using `PlayBuf`; you can create several outputs from it and generate a kind of impulse response with it. This is useful to simulate early reflections. (See figure 14.6.)

```
// Create a buffer.
b = Buffer.alloc(s, s.sampleRate, 1); //enough space for one   seco
nd of mono audio

// Write to the Buffer with BufWr, read using several taps and mix
 them together:
(
SynthDef(\helpTap, {|bufnum|
   var source, capture;
   source = Impulse.ar(1);
   capture = BufWr.ar(source, bufnum, Phasor.ar(0,1, 0,   BufFrame
s.ir(bufnum),1));
   Out.ar(0, Mix.new([1,0.95,0.94,0.93,0.8,0.4,0.4] *   Tap.ar(bufn
um, 1, [0.04,0.1,0.22,0.88,0.9,0.91,0.93])));
}).add;
)

x = Synth(\helpTap,[\bufnum, b.bufnum]);
x.free;

// alternate source; use headphones to avoid feedback
(
SynthDef(\helpTap2, {|bufnum|
   var source, capture;
   source = SoundIn.ar(0);
   capture = BufWr.ar(source, bufnum, Phasor.ar(0,1, 0,   BufFrame
s.ir(bufnum),1));
   Out.ar(0, Mix.new([1,0.95,0.94,0.93,0.8,0.4,0.4] *   Tap.ar(bufn
um, 1, [0.04,0.1,0.22,0.88,0.9,0.91,0.93])));
}).add;
)
```

```
x = Synth(\helpTap2,[\bufnum, b.bufnum]);
x.free;

// free buffer:
b.free;
```

Figure 14.6
`Tap` example.

`PingPong` bounces a signal between channels by doing the following: The input channels are recorded to a buffer; the buffer is played back with `delayTime`; that output is rotated with rotate channels (so all the channels shift their position); this is then multiplied by the feedback factor and in turn is added to the input to the recording buffer. The output that you hear is a summation of the input with the delayed signal after the rotation and the feedback multiplication.

14.4.5 Convolution

If you have a lot of delay taps, it is more efficient to use a `Convolution` UGen. In SuperCollider, there are several UGens available for convolution. Convolution allows you to convolve two running signals with one another; this can be useful for creating complex spectra of two different sounds. For spatialization the `Convolution2` UGens (`Convolution2`, `Convolution2L`, and `StereoConvolution2L`) are more useful since they use a buffer for the impulse response with which an input signal is convolved. `Convolution3` implements a time-based calculation of convolution; since this is a highly inefficient way of calculating a convolution, it is not recommended to use it at audio rate. The other `Convolution` UGens calculate the convolution in the frequency domain; that is, they calculate a Fourier transform of the input signal and of the kernel (either the other input signal or the impulse response buffer), and then multiply the coefficients before the output is transformed back into the time domain. This is computationally efficient, but it leads to significant latency with large impulse responses. `PartConv` implements real-time partitioned convolution, which addresses this problem by breaking the impulse response into smaller chunks. (See figure 14.7.)

```
// convolving two signals with each other:
Convolution.ar (in, kernel, framesize, mul, add)

// convolving one signal with a buffer:
Convolution2.ar (in, bufnum, trigger, framesize, mul, add)
// as above with linear interpolation:
Convolution2L.ar (in, bufnum, trigger, framesize, crossfade,   mul,
add)
// as above, with two buffers:
```

```
StereoConvolution2L.ar (in, bufnumL, bufnumR, trigger, framesize,
crossfade, mul, add)
// time based convolution (highly inefficient for audio rate)
Convolution3.ar (in, kernel, trigger, framesize, mul, add)
Convolution3.kr (in, kernel, trigger, framesize, mul, add)

// partitioned convolution
PartConv.ar(in, fftsize, irbufnum, mul, add)
```

Figure 14.7
The `Convolution` UGens and their argument.

14.4.6 Reverb

For reverb, you can use the `Comb` and `AllPass` UGens as shown above, or you can use a recorded impulse response and one of the `Convolution` UGens. Other options are using resonating filters or UGens such as `Decay`.

There are three UGens in the standard distribution designed specifically for reverb (see figure 14.8): `FreeVerb`, `FreeVerb2`, and `GVerb`. `FreeVerb` takes as parameters the dry/wet balance (called "mix"), a measure of the room size, and the amount of damping of the high frequencies; each of these parameters has a range of 0–1. `FreeVerb2` is a two-channel version that has two inputs and two outputs. `GVerb` gives a stereo output and has several more parameters to tune the reverb: room size (in meters), reverberation time (in seconds), damping (high-frequency roll-off, with a range of 0–1), input bandwidth (high-frequency roll-off on the input signal), amount of stereo spread and diffusion, the dry level, the amount of early reflections, the tail level, and the maximum room size.

```
// one channel input:
FreeVerb.ar(in, mix, room, damp, mul, add)
// 2 channel input and output:
FreeVerb2.ar(in1, in2, mix, room, damp, mul, add)

// stereo reverb
#left, right = GVerb.ar(in, roomsize, revtime, damping, inputbw,
spread, drylevel, earlyreflevel, taillevel, maxroomsize, mul, add)
```

Figure 14.8
The `Reverb` UGens with their input arguments.

Using different directions for reverb will enhance the feeling of envelopment; this is what GVerb does with two channels. For more channels, you can achieve this effect by using reverbs from several directions, with slightly different settings for each direction.

14.4.7 Arbitrary Spatialization

Depending on the project that you are working on, you may want to use a different kind of spatialization, considering the setup of the speakers that you have and the content of your work. As an example, we have included in the website materials a class created for spatialization of the sound of the dance theater piece *Schwelle* (Baalman, Moody-Grigsby, and Salter, 2007). For this piece, a setup was used which had an inner space and an outer space. The inner space was created by four speakers (two stereo pairs) surrounding the stage; the outer space, by speakers behind and above the audience. Various sound movements were defined within this space.

This example demonstrates one approach for creating a spatialization for a piece and how one might set up a framework for dealing with it. Similarly, in your own work, you can assign any algorithm to determine which speaker a sound is routed to and at what amplitude; this is your compositional freedom.

14.5 3D Audio

Three-dimensional (3D) audio encompasses all methods which try to re-create a 3D wave field in one way or another. This can be done either with headphone-based methods, such as binaural audio, or through loudspeaker methods, such as Ambisonics and wave field synthesis. In this section, we will discuss each of these methods and how they can be used within SuperCollider.

In all of these methods, there is a shift from a track- or channel-based approach to an object-oriented way of working. That is, instead of thinking about which sound should come from which channel, you think about the virtual position that the sound should have and feed the sound into the 3D audio system, along with the desired spatial coordinates and other spatial properties. From this metadata, the 3D audio system then calculates the appropriate signals for each channel, which will be fed to either headphones or loudspeakers.

14.5.1 Binaural Audio

Binaural methods attempt to present audio signals at each ear that correspond to the 3D audio field of an auralized scene. This is done by measuring the head-related transfer function (HRTF) for each ear, for all desired directions from which a sound source may come.[2] These HRTFs can be stored in a database as impulse responses. If we want to auralize a sound coming from a direction with an elevation of 30 degrees and an azimuth of 10 degrees, the corresponding HRTF (one impulse response for each ear) can be looked up, and the audio can be convolved with these impulse responses. The problem is actually a bit more complicated, as we can move our heads, and thus we need to change the HRTF not only when the source moves to another position, but also

when we turn our heads. For this, we need to have a device that measures our head position, called a “head tracker.”

When the auralized source is to appear to be in a room, HRTFs need to be created (measured or calculated) that include the acoustics of the room. This means that head and source position must be treated separately and an HRTF needs to be available for all of the following:

- Each sound source position
- Each listener position
- With each possible head position

The amount of data thus easily becomes very large and is organized across several dimensions. Depending on the requirements, the resulting data set may be impractically large for the RAM available and require a management strategy. The amount of data can be reduced by adapting the resolution: we can take an HRTF at coarser intervals and at fewer listening positions. This is, of course, a trade-off between data size and the accuracy of the binaural reproduction. Another limitation will be the number of sources that we can auralize simultaneously. For each source, we need to do 2 convolutions with each appropriate impulse response (and maybe even more, if we want to ensure that we can switch to another HRTF quickly), so our CPU power and the performance of the convolution algorithm will determine this limit.

The general structure of a binaural audio engine is shown in figure 14.9. The `SynthDef` for one binaural source is the following:

```
(
SynthDef(\binauralconvolver, {|out = 0, in = 0, bufL = 0,   bufR =
 1, t_trig = 0, amp = 1|
        Out.ar(out,
              StereoConvolution2L.ar(In.ar(in, 1), bufL, bufR,   t_
trig, 2048, 1, amp)
              // 2048 is the FIR size, 1 means that we crossfade ov
er 1 block between buffers
        );
}).add;
)
```

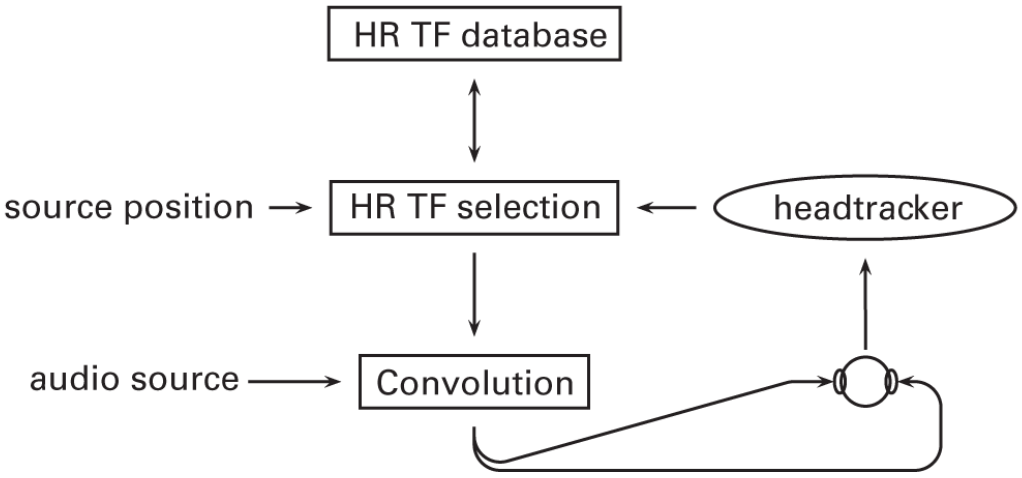


Figure 14.9
Binaural audio engine.

We can set the input bus with the argument in and the HRTF buffers with `bufL` and `bufR`. The HRTF buffers need to be chosen based on the head orientation and the source position. Suppose that we have a database with just angular positions relative to the head (azimuth and elevation); then we can create a matrix (`~HRTF`) which maps the azimuth and elevation angles to pairs of HRTF buffers.

Suppose that we have a head tracker attached to the headphones of the listener and data from it are streaming into SuperCollider. We have assigned the instance of the class dealing with the device to the variable `~headtracker`. Then we need to define an action to be performed when data change (we assume that the class has a method action for this, and that it takes the arguments `azimuth` and `elevation`). The source is represented in a class which contains the source position in the variables `azimuth` and `elevation` and is instantiated as `~source`; it also has an action that can be set to be done when the source position changes. The following code shows what this might look like:

```
x = Synth.new(\binauralconvolver, [\bufL, ~HRTF[~sourceazi]   [~sou
rcelev][0], \bufR, ~HRTF[~sourceazi][~sourcelev)[1]]);
// the buffers in ~HRTF are sorted at the first level by the   azim
uth, the second level by the elevation, and on the third   level by
left or right (the index 0 and 1 in the line above.
~headtracker.action_({|azim,elev| x.setn(\bufL, ~HRTF[~source.   az
imuth—azim].at[~source.elevation - elev]);});
// This function is executed when the headtracker provides   new in
formation and this checks the head's azimuth versus   the source's
 azimuth, and the head's elevation versus the   source's elevation.
We use the method .setn, which sets then   both arguments for \bufL
and \bufR. For this it is important   that in the binaural convolve
```

```
r, you also define these   arguments in the right order.
~source.action_({|azim,elev| x.setn(\bufL, ~HRTF[~azim -   ~headtra
cker.azimuth] [~elev - ~headtracker.elevation])});
// This function is executed when the source provides new   informa
tion and functions similar to the function for the   headtracker.
```

See Also External: IEM UGens

14.5.2 Vector Base Amplitude Panning (VBAP)

The Finnish researcher Ville Pulkki has developed an extension of pairwise panning techniques called Vector Base Amplitude Panning (Pulkki, 2001), which was ported to SC by Scott Wilson (2007) and is available as part of the sc3-plugins set of extensions. VBAP can be applied between pairs of loudspeakers (2D VBAP) or triplets (3D VBAP). As with basic stereophony, VBAP assumes that speakers are equidistant from the listener, so in the 3D case, this means a complete or partial dome or sphere.[3] A useful innovation in VBAP is that (in contrast to `PanAz`) the speakers do not have to be evenly spaced around the ring or dome. VBAP abstracts panning control from speaker configuration and allows accurate pairwise or triplet panning with arbitrary spacings, so it is easy, for instance, to move from an octophonic ring to a 5.1 setup. An algorithm determines the optimal pairs or triplets for a given setup. Panning position is indicated using 2 parameters expressed in degrees: azimuth, or horizontal pan, with 0 degrees directly to the front; and elevation (for 3D VBAP), expressed in degrees above or below the azimuth plane. Speaker positions are specified with the same parameters, which are used as input to the `VBAPSpeakerArray` object. Figure 14.10 presents examples of these.

```
// 5.1 array (subwoofer must be treated separately)
VBAPSpeakerArray.new(2, [-30, 30, 0, -110, 110]);

// 16 channel partial dome
VBAPSpeakerArray.new(3, [[-22.5, 14.97], [22.5, 14.97],   [-67.5, 1
4.97], [67.5, 14.97], [-112.5, 14.97], [112.5,   14.97], [-157.5, 1
4.97], [157.5, 14.97], [-45, 0], [45, 0],   [-90, 0], [90, 0], [-13
5, 0], [135, 0], [0, 0], [180, 0]]);
```

Figure 14.10
2D and 3D VBAP `Speaker Arrays`.

The `VBAP` UGen does the actual panning. In addition to the azimuth and elevation parameters, it has a spread parameter. When spread is equal to 0, a panned signal will be output from a single speaker if the pan azimuth and elevation exactly match that of a

speaker's location. Spread values greater than 0 (expressed as a percentage, with 100 indicating the entire array) ensure that this never happens by always having the signal played over at least two speakers. This can prevent signals from suddenly sounding "in the box" as they pan to an exact speaker position, by smoothing out the changes in "localization blur" that occur when panning between speakers (Pulkki, 1999). Values in the range of 10–20 have been found to be useful in informal testing.

Figure 14.11 gives an example of some 3D VBAP panning.

```
s.options.numOutputBusChannels = 16; s.boot;

a = VBAPSpeakerArray.new(3, [[-22.5, 14.97], [22.5, 14.97],   [-67.
5, 14.97], [67.5, 14.97], [-112.5, 14.97], [112.5,   14.97], [-157.
5, 14.97], [157.5, 14.97], [-45, 0], [45, 0],   [-90, 0], [90, 0],
 [-135, 0], [135, 0], [0, 0], [180, 0]]);   // zig zag partial dome

b = a.loadToBuffer; // send speaker config to the server

(
// pan around the circle moving up and down
x = {|azi = 0, ele = 0, spr = 10|
   var source;
   source = PinkNoise.ar(0.4);
   VBAP.ar(16, source, b, LFSaw.kr(0.5, 0).range(-180, 180) * -1,
SinOsc.kr(3, 0).range(0, 14.97), spr);
}.scope;
)
```

Figure 14.11
3D VBAP example.

14.5.3 Ambisonics

The Ambisonic sound system (Malham and Myatt, 1995) is a 2-part technological solution to the problem of encoding sound directions and amplitudes, and reproducing them over practical loudspeaker systems, so that listeners can perceive sounds located in a 3-dimensional space. This can occur only over a 360-degree horizontal sound stage (pantophonic system) or over the full or partial sphere or dome (periphonic system). The system encodes the signals in so-called B-format; the first-order version of this encodes the signal in three channels for pantophonic systems and a further channel for the periphonic (i.e., "with height" reproduction).

Essentially, the system gives an approximation of a wave field by a plane wave decomposition of the sound field at the listener's position. This approximation gets more precise when the order of Ambisonics is increased, which also means that more channels are needed for encoding and that more speakers are needed for decoding. Higher-order Ambisonics has become one of the most common ways to work with 3D spatial audio in recent years due to the availability of algorithms to easily create high-quality decoders for arbitrary speaker arrays.

There are two stages to an Ambisonics rendering: encoding the signal to the B-format, and then decoding it for the actual speaker layout. An encoded signal can also be manipulated before it is decoded, allowing you to rotate the sound field around different axes.

Within the main distribution of SuperCollider, there are a limited number of UGens supporting first-order Ambisonics.[4] Figure 14.12 gives an overview of these UGens and their arguments.

```
// 3D encoding:
PanB.ar(in, azimuth, elevation, gain)
// 2D encoding:
PanB2.kr(in, azimuth, gain)
// 2D encoding of a stereo signal:
BiPanB2.kr(inA, inB, azimuth, gain)

// decoding (2D):
DecodeB2.kr(numChans, w, x, y, orientation)

// rotating (in the horizontal plane):
Rotate2.kr(x, y, pos)
```

Figure 14.12
The Ambisonic UGens in the main SuperCollider distribution.

A number of external resources support more advanced Ambisonic work, including HOA based approaches in some cases. These include the JoshUGens from sc3-plug-ins, the AmbIEM and SC-HOA quarks, and the Ambisonics Toolkit (ATK) which is distributed in two parts (quark and sc3-plugins). The `VSTPlugin` UGen from IEM (https://git.iem.at/pd/vstplugin) provides another viable approach by allowing SC to host Ambisonic VSTs (see chapter 29), such as the IEM plug-in suite (https://plugins.iem.at).

14.5.4 Wave Field Synthesis

Wave Field Synthesis (WFS) was introduced in 1988 (Berkhout, 1988), and is an approach to spatial sound field reproduction based on the Huygens Principle. With WFS, it is possible to create a physical reproduction of a wave field; this has an advantage over other techniques in that there is no *sweet spot* (i.e., there is no point in the listening area where the reproduction is remarkably better than at other places); rather, there is a *sweet area,* which is quite large.

In comparison with Ambisonic techniques, wave field synthesis is better at reproducing spatial depth, though the vertical dimension which can be used in Ambisonics is lacking in most WFS implementations. For very high orders, Ambisonics can be equated to wave field synthesis (Daniel et al., 2003). A disadvantage to this approach is that you need many loudspeakers to get the desired effect, as well as appropriate multichannel sound cards and the necessary CPU power.

WFS is based on the Huygens principle, which states that a wave front can be synthesized by an infinite number of small sound sources whose waves will sum together (Huygens, 1690), as illustrated in figure 14.13. A listener then will not be able to determine the difference between a situation in which the wave front is real and one in which it is synthesized.

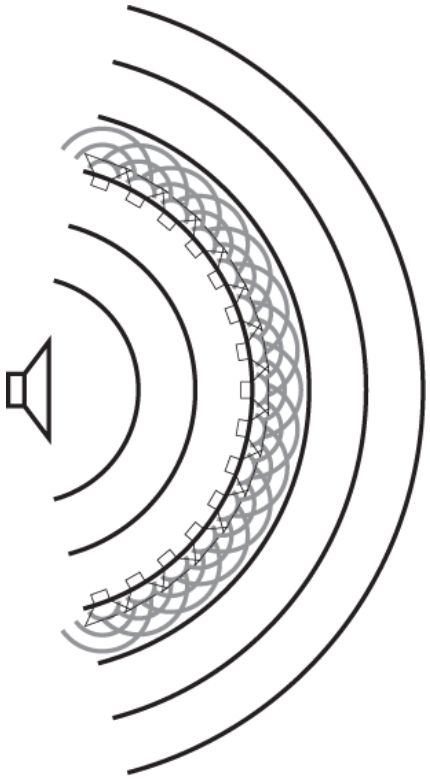


Figure 14.13
Huygens principle.

This principle can be translated to mathematical formulas using theories of Kirchhoff and Rayleigh and can then be applied for use with a linear array of loudspeakers (see, e.g., Baalman, 2008). By sending the correct signals to each loudspeaker (i.e., sending to the loudspeaker the signal that would be measured at the location of the speaker if the sound source would really have a given location), sound sources can be placed at any virtual place in a horizontal plane behind, or even in front of, the speakers (figure 14.14). Based on the desired virtual location for a sound source, you can calculate for each speaker in your speaker array the delay time, the amplitude factor, and a filter based on the coordinates of the speaker and of the virtual sound source, and a reference line in the listening area.

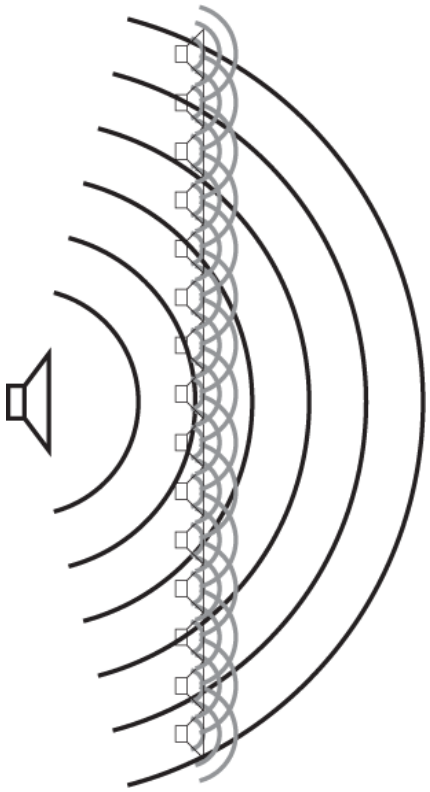


Figure 14.14
WFS.

In the Netherlands, the Game of Life Foundation sponsored the creation of a WFS system (see figure 14.15) implemented in SuperCollider and created by Wouter Snoei, Raviv Ganchrow, Jan Trützschler, and Miguel Negrão. In figure 14.16, you can see how the calculation for the delays and amplitudes is implemented in WFSCollider.[5]

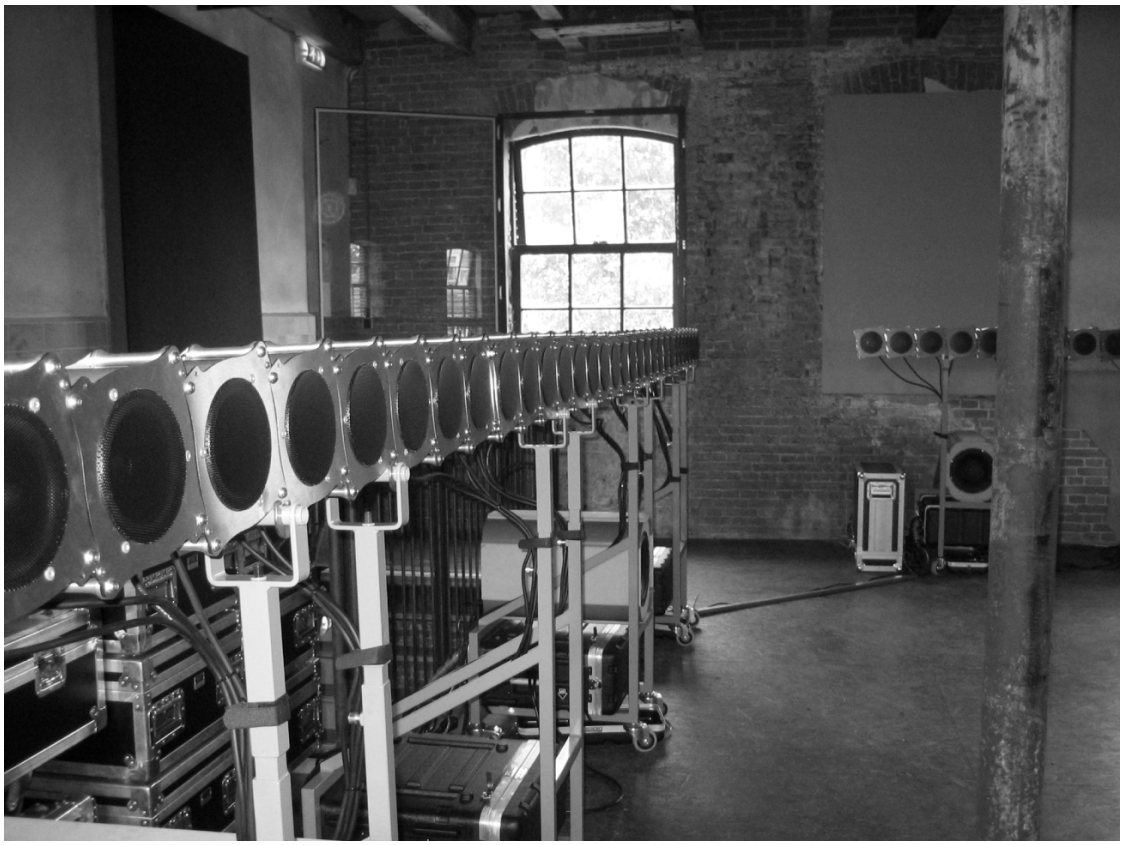


Figure 14.15
WFS speakers for the the Game of Life system (August 2007).

```
// no interpolation
*arBufN {arg sound = 0, bufnum, location, speakerSpec,   speedOfSou
nd = 334, ampType = 'ws';
var numChannels, distArray;

// WFSPoint is a 3D representation of a point in cartesian space

speakerSpec = speakerSpec ? [WFSPoint.new(-2.7, 1.8, 0),   WFSPoin
t.new(2.7,1.8, 0)]; //default 2 speakers

if(speakerSpec.class == WFSConfiguration)
   {speakerSpec = speakerSpec.allSpeakers;};

numChannels = speakerSpec.size; // speakerSpec = Array of   WFSPoin
t objects

location = location ? WFSPoint.new(0,0,0); // the location of   the
sound source
```

```
distArray = Array.fill(numChannels, {|i| speakerSpec.at(i)   .dist
(location)}); // distance of the sound source to each   speaker (r_
0)

cosPhiArray = Array.fill(numChannels, {|i| speakerSpec.at(i)   .cos
phi(location)}); // cosine phi_0 of the sound source to   each spea
ker (cos phi_0)
^BufDelayN.ar(bufnum, sound,
   distArray / speedOfSound, // delay
   WFSPan.wfsAmp(distArray,cosPhiArray);
}

*wfsAmp{arg inDist, inCosPhi, refDist=4.0, minDist = 0.1;
// refDist is the reference line distance
// minDist is the minimal distance to the speaker array (to avoid e
xplosion at /0)

// avoid explosion:
inDist = inDist.max(minDist);

^(ampFactor * ((refDist/(refDist + inDist)).sqrt)*(inCosPhi/   (inD
ist.sqrt)))
}
```

Figure 14.16
Example of WFS calculation from the WFS-lib from the `WFSPan` class by Wouter Snoei.

The tricky part comes as the WFS array gets larger, since at some point, one computer cannot manage all the calculations, and you need a technique for synchronizing several machines, which is implemented in WFSCollider.

The WFSCollider package also features an interface to define trajectories for sounds, sound-processing units, and scores to be played back on different configurations of WFS speaker Arrays.

14.6 Techniques for Spatial Diffusion (Decorrelation)

There are various spatial techniques that do not attempt to locate a sound precisely, but rather to make it more spatially diffuse (i.e., less localized), to broaden its sonic image, to increase its sense of physical volume, and so on. These usually work by taking a source signal and making decorrelated versions of it, which are distributed to two or more loudspeakers. As used here, the term “decorrelation” can be understood as the process of creating related yet dissimilar versions of a signal; that is, signals with

different wave forms that nevertheless sound very similar (Kendall, 1995). These effects have the advantage of being relatively stable across a wider variety of listening positions (i.e., a very large sweet spot), but all of them introduce artifacts of one sort or another.

It is possible to compare two signals in terms of their correlation and determine a correlation measure, which will be 1 for completely correlated signals and 0 for completely uncorrelated signals.[6] Consider the following 2 examples:

```
{WhiteNoise.ar(0.1)! 2} .play; // panned WhiteNoise sounds like
// it comes from the center
{{WhiteNoise.ar(0.1)}! 2} .play; // decorrelated channels of   // W
hiteNoise create an effect of width
```

The difference between the two examples is that in the first, the two channels play the same noise signal (so the correlation measure is 1), whereas in the second example, they play two different noise signals (so the correlation measure is close to 0). This effect is very obvious when you listen to the white noise directly, but it will still be apparent if you filter the sound afterward. For example:

```
{LPF.ar(Array.fill(2, WhiteNoise.ar(0.8)), 300, 0.2)}.play;
```

14.6.1 Phase Spectrum Decorrelation

The noise examples discussed here work because the two noise sources are *statistically* (and sonically) essentially the same, but decorrelated. When using general signals, you can create decorrelated variants by altering the phase spectrum of a signal, using techniques such as those described by Kendall (1995). This involves applying a random phase offset to each bin of an FFT analysis. The standard distribution of SuperCollider contains the `PV_Diffuser` UGen, and as part of the BEASTmulch project at the University of Birmingham, Scott Wilson has developed a variant of this called `PV_Decorrelate` (Wilson, 2007). `PV_Decorrelate` has an additional scaling factor applied to the random offsets, effectively allowing the maximum offset to be limited. Setting this argument to 0 will result in completely correlated signals (no phase randomization), and setting it to 1 should result in maximum phase randomization (and a correlation measure close to 0). Controlling this is desirable in some cases since the phase offsets can introduce audible artifacts such as transient smearing. Figure 14.17 demonstrates the use of `PV_Decorrelate`. Moving the mouse to the right increases the amount of decorrelation, creating a more spatially diffuse stereo effect but also adding phasing artifacts.

```
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");

(
// make stereo from mono
// MouseX controls decorrelation
x = {
var in, chain, chain2;
in = PlayBuf.ar(1, b, BufRateScale.kr(b), loop: 1);
   chain = FFT(LocalBuf(2048, 1), in);
chain = PV_Decorrelate([chain, chain], 1, MouseX.kr);
Out.ar(0, 0.5 * IFFT(chain));
}.play
)

x.free; b.free;
```

Figure 14.17
An example of the use of `PV_Decorrelate`.

14.6.2 Granular Techniques

The term "granulation" refers to a group of related techniques which involve breaking a sound into a small number of very short, usually overlapping, pieces. This has applications for time stretching, pitch shifting, brassage, and so on. A detailed discussion of this rather extensive topic is beyond the scope of this chapter, but for general information on granulation, see Roads (2001) and Truax (1990). For information on granulation in SC, see chapter 16.

It is possible to create diffusion effects by granulating over a number of channels. Individual grains can either be panned between channels or hard assigned.[7] SC contains a number of UGens for granulating sound, such as `TGrains`, `GrainBuf`, and `GrainIn`; all of them support multichannel output. Figure 14.18 presents an example of a client-side approach for smooth spatial granulation. Granulation procedures are not without audible artifacts (increasing the number of grains can help to smooth this out, but at the cost of increased CPU load), but they are nevertheless capable of producing a number of interesting effects, spatial and otherwise.

```
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");
(
SynthDef(\grainWithSineEnv, {|rate = 1, sustain = 1, amp = 0.1,   p
os = 0, out = 0, buf| // buf is an argument here
   var pb;
   // mono playbuf
```

```
    pb = PlayBuf.ar(1, buf, rate * BufRateScale.kr(buf),   startPos:
pos, loop:1);
    pb = pb * EnvGen.ar(Env.sine(sustain, amp), doneAction:2);
    OffsetOut.ar(out, pb); // offset out for precise timing
}).add;
)

// granulate with 2X time stretch
(
var numChannels = 2; // adjust for bigger setups
Pbind(
    \instrument, \grainWithSineEnv,
    \sustain, Pwhite(0.05, 0.1), // grain duration
    \delta, Pkey(\sustain) / 8, // approximately 8 simultaneous   gr
ains
    \pos, Pseg([0, b.numFrames], [b.duration * 2, 0]),   // position
of grain in the buffer
    \out, Prand((0..(numChannels-1)), inf), // random hard pan
    \buf, b // pass the buffer here
).play;
)
```

Figure 14.18
Spatial diffusion through granulation.

14.6.3 Spectral Diffusion

It is possible to distribute the spectral content of a sound spatially across a number of loudspeakers. Such techniques arguably have their roots in the analog "spectral splitting" effects of loudspeaker orchestras such as the Gmebaphone/Cybernéphone of the Institut international de musique électroacoustique de Bourges (IMEB), the Acousmonium of the Groupe de Recherches Musicales (GRM), and Birmingham ElectroAcoustic Sound Theatre (BEAST). These systems, either intentionally or implicitly, use filtering and nonhomogeneous loudspeaker arrays to increase the diffuseness of sounds through spectral distribution in space.[8]

Similar effects can be created digitally through FFT-based analysis and resynthesis techniques. These approaches can be divided roughly according to the resynthesis technique used: Overlap-Add versus Oscillator Bank. The former are more straightforward in real-time and generally less computationally expensive as well. The latter are generally more robust, however, and result in fewer artifacts when subjected to transformation.

These effects work best when they occur over larger numbers of channels (i.e., preferably more than stereo). The effect can be subtle depending on the material, but it can be very effective at creating an impression of envelopment, sometimes rendering sounds impossible to localize.

Figure 14.19 presents an example of an Overlap-Add approach. As in the phase decorrelation examples discussed above, this approach creates decorrelated variants of a signal, in this case by dividing the energy in the magnitude spectrum between copies.

```
Server.default = s = Server.internal;
s.boot;
(
n = 512; // number of bins

// create arrays of magnitude scalars and load them to buffers
~leftScals = Array.fill(n, {1.0.linrand});
~rightScals = 1.0-~leftScals;
~scalBuf1 = Buffer.loadCollection(s, ~leftScals);
~scalBuf2 = Buffer.loadCollection(s, ~rightScals);

~soundBuf = Buffer.read(s, Platform.resourceDir +/+ "sounds/   a11w
lk01.wav");
)

(
x = {
   var chain;
   // two channels of FFT
   chain = FFT(LocalBuf([n, n]), PlayBuf.ar(1, ~soundBuf,   BufRate
Scale.kr(~soundBuf), loop: 1)! 2);
   //chain = FFT(LocalBuf([n, n]), PinkNoise.ar(0.3)! 2);   // swap
in for alternate source sound
   chain = PV_MagMul(chain, [~scalBuf1, ~scalBuf2]);
   0.5 * IFFT(chain);
}.play;
s.scope; // compare the two channels
)

// execute this multiple times to change the distribution
(
~leftScals = Array.fill(n, {1.0.linrand});
~rightScals = 1-~leftScals;
~scalBuf1.loadCollection(~leftScals);
```

```
~scalBuf2.loadCollection(~rightScals);
)
x.free; [~scalBuf1, ~scalBuf2].do(_.free); // cleanup
```

Figure 14.19
Simple spectral diffusion example.

Techniques which use Oscillator Bank resynthesis generally require that the analysis be done in advance, and thus they are not suitable for use with real-time input. Two such approaches have SC-based implementations: Juan Pampin's ATS (Pampin 1999; SC port by Joshua Parmenter) and Kelly Fitz and Lippold Hakken's Loris (Fitz et al., 2002; SC port by Scott Wilson, 2007). Both of these provide the user with a list of partials, each of which can be individually spatialized when resynthesized using any of the techniques described in this chapter. An example of spectral diffusion using Loris can be found in the website materials.

14.7 Conclusion

As you hopefully have seen from the above examples, considering the aspect of space in your work with SuperCollider can add richness, depth, and realism. Some of the approaches presented in this chapter work best on large numbers of speakers, but even in stereo on a laptop, you can do a lot. Why not take some of the other examples in this book and alter or adapt them using some of the techniques we have discussed? As we noted at the start of this chapter, sounds always have a spatial aspect in the real world; there is no reason why your SC sounds should not do so as well, regardless of how synthetic their origins might be.

Notes

1. Note that some cinema formats have speaker placements in which angles of greater than 60 degrees subtend the listening position, which can cause audible gaps in the panning effect. This is not a design flaw, as the surround speakers are intended to provide ambience in cinema playback, not enable precise localization of point sources to the rear of the listener.
2. As noted earlier, the filtering effects resulting from reflections off the ridges of the ears—as well as other parts of the body, such as the shoulders and chest—play a crucial role in spatial hearing. Head-related transfer functions model this mathematically and allow one to simulate this effect for a given position.
3. One can compensate for other shapes, such as cylinders, by applying appropriate delays to the closer speakers so the precedence effect does not come into play.
4. more are available in sc3-plugins: https://github.com/supercollider/sc3-plugins
5. https://github.com/GameOfLife/WFSCollider-Class-Library.
6. The technical term for this process is the "cross-correlation function." See Kendall (1995) for a discussion of this topic.
7. The perceptual differences between panned and hard assigned grains are very small when using a reasonable number of short concurrent grains over more than two channels.
8. See Clozier (2001), Harrison (1999), and Tutschku (n.d.) for further discussion.

References

Ando, Y. 1985. *Concert Hall Acoustics*. Springer Series in Electrophysics, Vol. 17. Berlin: Springer.

Baalman, M. A. J. 2008. "On Wave Field Synthesis and Electro-Acoustic Music, with a Particular Focus on the Reproduction of Arbitrarily Shaped Sound Sources." PhD thesis, Technical University of Berlin.

Baalman, M. A. J. 2010. "Spatial Composition Techniques and Sound Spatialisation Technologies." *Organised Sound,* 15(3): 209–218.

Baalman, M. A. J., T. Hohn, S. Schampijer, and T. Koch. 2007. "Renewed Architecture of the sWONDER Software for Wave Field Synthesis on Large-Scale Systems." In *Proceedings of the 5th Linux Audio Conference 2007,* pp. 76–83. Berlin: Technical Universität.

Baalman, M. A. J., D. Moody-Grigsby, and C. Salter. 2007. "Schwelle: Sensor Augmented, Adaptive Sound Design for Live Theater Performance." In *Proceedings of the 7th International Conference on New Interfaces for Musical Expression,* pp. 178–184. New York: New York University.

Berkhout, A. J. 1988. "A Holographic Approach to Acoustic Control." *Journal of the Audio Engineering Society*, 36(12): 977–995.

Blauert, J. 1997. *Spatial Hearing,* rev. ed. Cambridge, MA: MIT Press.

Clozier, C. 2001. "The Gmebaphone Concept and the Cybernéphone Instrument." *Computer Music Journal*, 25(4): 81–90.

Daniel, J., R. Nicol, and S. Moreau. 2003. "Further Investigations of High Order Ambisonics and Wavefield Synthesis for Holophonic Sound Imaging." In *Proceedings of the 114th Convention of the Audio Engineering Society.* Convention preprint 5825.

Fitz, K., L. Haken, S. Lefvert, and M. O'Donnell. 2002. "Sound Morphing Using Loris and the Reassigned Bandwdith-Enhanced Additive Sound Model: Practice and Applications." In *Proceedings of the 2002 International Computer Music Conference.* Ann Arbor: Scholarly Publishing Office, University of Michigan Library.

Harrison, J. 1998. "Sound, Space, Sculpture: Some Thoughts on the 'What,' 'How' and 'Why' of Sound Diffusion." *Organised Sound*, 3(2): 117–127.

Huygens, C. 1690. *Traité de la lumière; Où sont expliquées les causes de ce qui lui arrive dans la réflexion et dans la réfraction et particulièrement dans l'étrange refraction du cristal d'Islande; avec un discours de la cause de la pesanteur*. Leiden, Netherlands: Pieter van der Aa.

Kendall, G. 1995. "The Decorrelation of Audio Signals and Its Impact on Spatial Imagery." *Computer Music Journal*, 19(4): 71–87.

Malham, D. G., and A. Myatt. 1995. "3D Sound Spatialization Using Ambisonic Techniques." *Computer Music Journal*, 19(4): 58–70.

Pampin, J. 1999. "ATS: A Lisp Environment for Spectral Modeling." In *Proceedings of the 1999 International Computer Music Conference.* Ann Arbor: Scholarly Publishing Office, University of Michigan Library.

Pulkki, V. 1999. "Uniform Spreading of Amplitude Panned Virtual Sources." In *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics,* pp. 187–190.

Pulkki, V. 2001. *Spatial Sound Generation and Perception by Amplitude Panning Techniques.* Technical Report 62. Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology.

Roads, C. 2001. *Microsound.* Cambridge, MA: MIT Press.

Rumsey, F. 2001. *Spatial Audio*. Oxford, UK: Focal Press.

Sonke, J.-J., and D. de Vries. 1997. "Generation of Diffuse Reverberation by Plane Wave Synthesis." Preprint 4455. In *Proceedings of the 102nd AES Convention.* Munich: AES.

Truax, B. 1990. "Time Shifting of Sampled Sound with a Real-Time Granulation Technique." In *Proceedings of the 1990 International Computer Music Conference.* San Francisco: ICMA.

Tutschku, H. n.d. "On the Interpretation of Multi-channel Electroacoustic Works on Loudspeaker-Orchestras: Some Thoughts on the grm-acousmonium and BEAST." Translated by George Goodman. http://www.tutschku.com/content/interpretation.en.php.

Wilson, S. 2007. BEASTmulch. University of Birmingham, https://www.birmingham.ac.uk/facilities/ea-studios/research/mulch.aspx.

15 Machine Listening in SuperCollider

Nick Collins

15.1 Background and First Steps

15.1.1 What Is Machine Listening?

Machine listening is the capability of machines to simulate human auditory and musical abilities. Why would we want to forge that talent in machines? Well, imagine being the poor musician who has to play along with a prerecorded tape part or click track. Wouldn't it be more natural to play with an accompanist who can adapt in the event of a mistake or a change of plan? What if you wished to engage in improvisational dialogue, sparring with an artificial musician, or to interact with a sound installation via your voice? Computers can provide remarkable generative and processing powers, but they are integrated so much more naturally if they are supportive of human mechanisms of interaction. Machine listening gives a way to equip the computer for those social musical and sonic circumstances which we take for granted when resting on our own cultural training and biological heritage.

Nevertheless, it is a tall order to match millions of years of evolutionary fitness testing in sounding environments and the frantic pace of cultural memetics with a computer program. It is best for me to confess at the outset that the state of the art in machine listening falls short of human auditory acuity, particularly in tracking multiple simultaneous objects in an auditory scene or extracting high-level, culturally contingent musical information. In many cases, the proof that a certain ability is possible rests with human beings alone and has not been successfully matched by machines. Nevertheless, I hope this chapter will demonstrate that we can engineer useful aural and musical abilities, and indeed do so within our SuperCollider patches. As the subject of intensive research efforts, enhancements will continue to be released in the years to come, but readers will no doubt be eager to see what can be done right now.

Machine listening in SuperCollider operates causally in real time, allowing the recognition of auditory objects and the tracking of features such as fundamental frequency, loudness, timbre, and event onset, as well as musical constructs such as beat, key, and phrase. In this chapter, I will describe the various UGens and classes in SuperCollider that support such abilities, in turn enabling real-time interactive music

systems. It will gradually progress according to the complexity of the tools, ending with a discussion of interactive music systems that potentially combine many machine listening UGens with higher-level compositional decisions.

15.1.2 A Practical Example

The SuperCollider patch in figure 15.1 combines the `Pitch` and `Amplitude` UGens to let you control a triangle waveform with your voice; you might want to try it wearing headphones in case of any feedback. The `Pitch` UGen is continuously extracting an estimate of the fundamental frequency, and `Amplitude` is continually tracking the amplitude level of your voice. These parameters are then used to drive a triangle oscillator, though they could very well be controlling a massed bank of oscillators, playback of someone else's prerecorded voice, or an arbitrarily strange network of UGens. Both UGens operate in the time domain, and we'll explore both time and frequency domain processes in this chapter. You won't often need to worry about this distinction, since UGens will be doing the hard work for you, but you need to be aware that for frequency domain processes, we might use the FFT UGens as a first step in real-time analysis; other examples in this chapter will make this clear.

```
(
x = {
    var in, amp, freq, hasFreq, out;
    in = SoundIn.ar(0);
    amp = Amplitude.ar(in);
    # freq, hasFreq = Pitch.kr(in);
    LFTri.ar(freq*[1,2]) * amp;
}.play
)
x.free;
```

Figure 15.1
Immediate machine listening example using `Pitch` and `Amplitude` UGens. The original detected pitch appears in your left ear and an octave higher in the right one.

The `Amplitude` UGen tracks the amplitude of a signal—typically in the range −1.0 to 1.0 for floating-point audio, where 0 is silence and 1.0 is the absolute maximum output level of the sound card—smoothed over a short time period. The amount of smoothing is determined by the attack and release time parameters, as further discussed in the Help file.

The `Pitch` UGen is more complicated and has quite a few arguments, which you'll see if you check the code or Help file:

```
Pitch.kr(in = 0.0, initFreq = 440.0, minFreq = 60.0, maxFreq =   40
00.0, execFreq = 100.0, maxBinsPerOctave = 16, median = 1,   ampThr
eshold = 0.01, peakThreshold = 0.5, downSample = 1)
```

In practice, you can often accept the defaults, and that goes for many of the UGens described in this chapter. As you become more expert, you'll naturally end up tweaking things; no machine listening UGen is perfect for all circumstances, and it's great to have a wide selection of analyzers and feature extractors to choose among.

The readers may be concerned by the 2 outputs from the `Pitch` UGen, which are (1) the current fundamental frequency estimate in hertz, and (2) a measure of whether a pitch was detected in the signal, as a simple flag of 1 for yes and 0 otherwise. The `#freq, hasFreq` construction in the code deals with assigning the two outputs to two separate variables for ease of reuse; otherwise, the output would be an array of two elements. We'll see multiple outputs many times in the following discussion.

How does the `Pitch` UGen work? Though we're not going to dwell on every detail of the mechanisms of the machine listening UGens in this chapter (there isn't space, and the average user probably doesn't need to know about that anyway), I'll try to drop a few hints to get the curious reader started. Pitch extracts fundamental frequency, often written as *f0* in technical literature; *f0* denotes the base partial of a harmonic series, although the phenomenon of the missing fundamental in psychoacoustics means that this partial does not have to be physically present in order for that pitch to be heard. The UGen implements a time-domain autocorrelation-based *f0* follower, which means that it measures at what spacing in time a shifted version of the waveform best lines up with itself. It tries a number of possible shift sizes, called "lags," which essentially reveal the period of the waveform. Unfortunately, there are a few ambiguities; the waveform will also line up well at double the period, causing a potential error of an octave. Indeed, a host of different pitch extraction algorithms have been devised, all with various possible confounds. But to get you started, the `Pitch` UGen in SuperCollider is a good *f0* tracker for most circumstances. If you're curious about this theory, I heartily recommend Alain de Cheveigné (2006).

15.1.3 The Information Which Machine-Listening Plug-ins Provide

Let's pause at this stage to consider two practical questions that may be preying on the minds of readers in the context of the last section:

1. What if I need discrete musical notes rather than continuous frequency information?
2. How do I get the outputs from a `Pitch` UGen back to the language so I can make compositional decisions based on them?

Though I've written both questions as they pertain to the pitch following UGen that we just used, these are actually more general questions that may occur for any feature extraction process.

In answer to these points in turn, the first draws our attention to the status of different time scales of activity; certain structures are convenient for describing the auditory information at each level of granularity. The lowest level is the smallest grain of individual samples, effectively continuous for frequencies below the Nyquist frequency. Multiple consecutive samples can be combined in a short-term window (also called a "frame"), perhaps of around 1–100 milliseconds, and a particular single value, a feature, obtained as a descriptor of some meaningful property (such as pitch or amplitude) of the audio within that window. A compression of data and of sampling rate is necessarily effected. Such a process typically operates at the level of multiple blocks; a standard window size is that of an FFT frame, such as 1,024 samples (16 sample blocks of 64 each, at around 23 ms for a 44.1-kHz sampling rate). A higher level combines frames, describing longer-scale sound objects that are intuitively sonic events, called "clangs" (Tenney, 2006), or musical notes; features might transfer to this level by taking a statistic over the object in question, such as a peak power, a median pitch, or the mean of some other feature value such as the spectral centroid. The combination or apportioning of these to phrases, measures, sections, or indeed whole works proceeds in turn.

Segmentation is a process which partitions an audio stream into more discrete musical objects, though again we may identify such salient events at a number of time scales (Roads, 2001). Although Trevor Wishart (1985), among others, has decried "lattice" thinking and the dominance of notation over pure experiential gesture, a categorization giving a small set of permissible "objects" for ease of use is a standard tactic in human endeavor and a necessity in many algorithms; we acknowledge, however, the multiple time scales and representations of objects over which audio analysis can operate.

In the familiar MIDI representation, the note objects have already been identified (though their "offs" may have been separated from their "ons," but that's another story). MIDI is an example of a clearly symbolic paradigm, and though this granularity of information can be helpful to us, it has also dropped certain fine-scale information on temporal variation. In considering interactive music systems at the end of this chapter, I'll return to MIDI and discuss an example of an interactive music system which is inherently symbolic and entirely language-side in its decision-making.

The second of our questions is a practical matter of getting information from the synthesis server, where the machine listening UGens are running, back to the language. There are a number of ways to do this, and different UGens may promote different methods. The possibilities are polling control buses (including shared buses on the

internal server) or buffers on the server from the language, and UGens (such as `SendTrig` and `SendReply`), which send OSC messages from the server back to the language. The code examples accompanying this chapter demonstrate these approaches.

Briefly, the server is the location for low-level signal processing, starting from an audio signal. The language is a better location for higher-level musical decisions, that is, *symbolic* processing. When making design decisions on your code, you might want to ponder how electronic music composition lets you make compositional decisions at a number of scales. Further illustrations of these issues will occur in later examples.

15.2 Features and Segmentation

15.2.1 Feature Extraction

So, we've already seen two UGens which, respectively, extract fundamental frequency and amplitude. What other features might we extract? Well, audio signal-processing engineers have discovered many interesting features which can be obtained from signals, all of them potentially useful for artistic purposes. Though those features that are most analogous to the capabilities of our own hearing systems are often more powerful, any abstract feature obtained from a signal might prove valuable in art. However, the bias of this chapter is toward machine listening, which is close to human listening.

There is some distinction between relatively low-level features and higher-level features. The progression of this chapter is to gradually move upstream. The convention is to associate *higher = more discrete = summarizing larger time scales* and *lower = more continuous = closer to an audio signal itself*.

Table 15.1 lists a selection of features which may be extracted and the corresponding SuperCollider UGens. Many are in the core of SuperCollider, but there are also some third-party UGens and classes; their author and project name are revealed as necessary. New UGens become available over time, and this list is by no means exhaustive; but I hope this chapter will still provide a useful starting point to the reader in exploring these possibilities. As is normal for third-party UGens, you will need to add the binaries, class, and Help files to your platform-specific SuperCollider Extension folder; instructions on completing this process are available in other chapters in this book, in the Using Extensions Help file, and in the README files that come with such downloads.

Table 15.1
Overview of machine listening resources for SuperCollider

Category	UGen/Project	Description	Availability (core unless otherwise stated)

Category	UGen/Project	Description	Availability (core unless otherwise stated)
Amplitude	`Amplitude`		
	`RunningSum`	RMS amplitude	
	`Loudness`	Perceptual loudness	
Pitch	`ZeroCrossing`	Simplistic pitch detection; also as a timbral feature	
	`Pitch`	Autocorrelation in time domain	
	`Tartini`	Adaptation from Phil McLeod's Tartini program	sc3-plugins/PitchDetection
	`Qitch`	Constant Q transform pitch detector (after Judith Brown and Miller Puckette)	sc3-plugins/PitchDetection
	`PolyPitch`	Polyphonic pitch detection algorithm; can look for more than one active pitch stream (after the work of Anssi Klapuri)	https://github.com/musicmichaelc/PolyPitch
	`Chromagram`	Track energy in a set of user-defined "chroma" bands (e.g., in 12TET, all C's, all C#s, etc.)	sc3-plugins/SCMIRUGens
Timbre	`SpecCentroid`, `SpecFlatness`, `SpecPcile`	Spectral features correlating with perceived brightness, noisiness, and rolloff point, respectively	
	`MFCC`	Mel frequency cepstral coefficients	
Onset detection	`PV_HainsworthFoote`		

Category	UGen/Project	Description	Availability (core unless otherwise stated)
	PV_JensenAndersen		
	Onsets	Dan Stowell's onset detector encapsulates his research work, including many different onset functions (Stowell and Plumbley 2007)	
	Coyote	UGen from Batuhan Bozkurt	sc3-plugins/ BatUGens
Event extraction	AnalyseEvents2		sc3-plugins/ BBCut2UGens
Beat tracking	BeatTrack	Autocorrelation beat tracker	
	BeatTrack2	Crosscorrelation beat tracker	
	DrumTrack	Drum pattern template detector	sc3-plugins/ BBCut2UGens
Key extraction	KeyTrack		
Psychoacoustics	DissonanceLib classes	Language side classes for sensory dissonance analysis by Juan Sebastian Lach	Quarks
Music information retrieval	SCMIR	Multiple UGens and language side resources for NRT and live audio content analysis (Collins 2011)	https://github.com/sicklincoln/SCMIR
	FluCoMa	General signal-processing library with many audio analysis capabilities (Tremblay, Roma and Green 2021)	https://www.flucoma.org/

Category	UGen/Project	Description	Availability (core unless otherwise stated)
Auditory modeling	`Gammatone, HairCell, Meddis`	Inspired by physiological models of human auditory function	sc3-plugins/ AuditoryModeling
Sparse representations	`MatchingP, MatchingPResynth`	Matching pursuit algorithm to analyze sound in terms of a user-provided dictionary of "atoms"; implemented as UGens by Dan Stowell	sc3-plugins/MCLDUGens
Essentia library	`EssentiaHFC,EssentiaSpectralComplexity`	UGens by Jildert Viet adapted from the Essentia library (Bogdanov et al. 2013)	https://github.com/jildertviet/SuperColliderEssentiaUGens

As an example, we will next consider extracting two psychoacoustically motivated features.

The first is a measure of perceptual loudness (Moore, 2004; Zwicker and Fastl, 1993) which utilizes a simple auditory model, using equal loudness contours and spectral and temporal masking. The output of the `Loudness` UGen, in the psychoacoustic unit of sones, is a better measure of the human cognition of volume than the `Amplitude` UGen, which treats only a physical (as opposed to perceptual) definition of the signal level. Figure 15.2 demonstrates the extraction of perceptual loudness, using the `Loudness` UGen on an audio input.

```
b = Buffer.alloc(s,1024,1); //for sampling rates 44100 and 48000
//b = Buffer.alloc(s,2048,1); //for sampling rates 88200 and 96000
(//analyse loudness and poll result
x = {
    var in, fft, loudness;
    in = SoundIn.ar(0);
    fft = FFT(b.bufnum, in);
    loudness = Loudness.kr(fft);
```

```
    loudness.poll(20); //poll for testing 20 times per second
    Out.ar(0,Pan2.ar(in));
}.play
)

x.free;
b.free;
```

Figure 15.2
Loudness.

One important factor to note concerning some machine listening UGens is that they are highly sample-rate-dependent. The `Loudness` UGen supports four sample rates: 44,100 and 48,000, using a 1,024-point FFT; and 88,200 and 96,000, using a 2,048-point FFT. Such issues are noted in the associated Help files, and readers should take care when running patches that they are aware of the server sample rate (`s.sampleRate`), which will match that of their assigned sound card.

The second psychoacoustic feature is actually a set of features rather than a single quantity. The Mel Frequency Cepstral Coefficients (MFCC; Logan, 2000) are often used to measure timbre in speech recognition and music information retrieval. In this case, we can choose how many coefficients to extract, getting one output channel (feature stream) for each coefficient. In technical terms, the coefficients are the result of matching a cosine basis to a frequency-warped spectrum. The `MFCC` UGen uses the Mel scale, a common psychoacoustic scale that models auditory sensitivity on the basilar membrane. (Other work might use critical bands in Barks or ERBs.) The cosine transform which takes us from a spectrum to a Mel Cepstrum is really just an approximation to principal component analysis that allows a dimensionality reduction, that is, finding the most important summary features for the spectral content of the signal. A side effect of MFCC analysis is that it separates the excitation from the body filter of an instrument, thus further confirming the measurement of timbre.

Figure 15.3 demonstrates obtaining 13 MFCCs with the `MFCC` UGen and observing those values from the language.

```
b = Buffer.alloc(s,1024,1); //for sampling rates 44100 and 48000
//b = Buffer.alloc(s,2048,1); //for sampling rates 88200 and 96000
//d = Buffer.read(s,"sounds/a11wlk01.wav");
(
x = {
    var in, fft, array;
    //in= PlayBuf.ar(1,d.bufnum,BufRateScale.kr(d.bufnum),1,0,1);
    in = SoundIn.ar(0);
```

```
    fft = FFT(b.bufnum, in);
    array = MFCC.kr(fft);
    array.size.postln;
    Out.kr(0,array);
    Out.ar(0,Pan2.ar(in));
}.play
)

c= Bus.new('control', 0, 13);
//poll coefficients
c.getn(13,{arg val; {val.plot;}.defer});
//Continuous graphical display of MFCC values; free routine // befo
re closing window
(
var ms;
w = Window("Thirteen MFCC coefficients", Rect(200,400,300,300));
ms = MultiSliderView(w, Rect(10,10,260,280));
ms.value_(Array.fill(13,0.0));
ms.valueThumbSize_(20.0);
ms.indexThumbSize_(20.0);
ms.gap_(0);
w.front;
r = {
    inf.do{
        c.getn(13,{arg val; {ms.value_(val*0.9)}.defer});
        0.04.wait; //25 frames per second
    };
}.fork;
)
//tidy up
(
r.stop;
b.free;
c.free;
x.free;
w.close;
)
```

Figure 15.3
MFCC.

15.2.2 Onset Detection

An important step between continuous and discrete auditory representation is the extraction of sound objects (typically 100–500 ms long; you might like to imagine them as "notes" or "events"), a process often dubbed *onset detection* or *segmentation*. We have to be careful about the kind of audio signal that we consider here: polyphonic audio, with many simultaneous parts, or complex auditory scenes, such as cocktail parties, require additional treatment in grouping information both vertically in frequency and horizontally across time. It is much easier to consider the case of monophonic audio with a solo instrument or speaker. Indeed, true polyphonic extraction in real time is beyond the cutting edge of audio processing, and in many cases, it is not a well-defined operation, so I make no apologies for downplaying it here.

There are various UGens in the SuperCollider core which can be used for onset detection, some specialized and some which can be adapted and combined into detectors. Starting from the observation that new events often begin with a transient burst of energy, it is instructive to try to build onset detectors using the combination of some simple UGens, such as `Amplitude`, `<`, and `Trig` (consider `Trig1.kr(Amplitude.kr(input) > 0.5)`, for example). The reader who tries this may quickly discover that it is not necessarily as simple a process as it might intuitively appear; not all note boundaries show up as changes in signal amplitude, and even if energy detection is appropriate, the right threshold and time scale to employ are tricky questions. In general, the better onset detectors are specialist UGens devised by researchers (`PV_HainsworthFoote` and `PV_JensenAndersen` are 2 examples).

Dan Stowell has released an easy-to-use collection of onset detectors as a by-product of his research into real-time vocal processing (Stowell and Plumbley, 2007). These are now encapsulated in the `Onsets` UGen in the standard distribution. Figure 15.4 gives an example of using the UGen with a particular choice of *detection function* from among the various options available. The code shows how the `SendTrig` UGen can be used to send a message back to the language when an onset is detected.

Essentially, onset detection is most effective for percussive signals. Soft onsets, such as slow attacks and subtle changes marked in pitch, vibrato, or multiple feature combinations, are more troublesome; you will need to try different solutions (UGens and their arguments, particularly thresholds and FFT settings) to find the best-performing detectors for the singing voice or nonpercussive instruments that go with particular microphones and acoustics.

```
// Prepare the buffer
b = Buffer.alloc(s, 512);

(
x = {
```

```
    var sig, chain, onsets, pips, trigger;
    sig = SoundIn.ar(0);
    chain = FFT(b, sig);
    //—move the mouse left/right to change the threshold:
    onsets = Onsets.kr(chain, MouseX.kr(0,1), \complex);
    trigger= SendTrig.kr(onsets);
    pips = SinOsc.ar(880, 0, EnvGen.kr(Env.perc(0.001, 0.1, 0.2),
 onsets));
    Out.ar(0, ((sig * 0.1) + pips).dup);
}.play;
)

// register to receive message
OSCdef(\receiveonset, {|msg, time, addr, recvPort|   [\onset,time].
postln}, '/tr', s.addr);

OSCdef(\receiveonset).free //Free the OSC callback
x.free; // Free the synth
b.free; // Free the buffer
```

Figure 15.4
Onsets.

15.3 Higher-Level Musical Constructs

We proceed now to a higher-level viewpoint and consider the extraction of what are more definite musical parameters from the perspective of conventional Western music theories. While we keep an ear to the wider currents of music in the world, the musical structures that we now seek to extract are related to common-practice notions of meter and key, and they assume particular theories on the perception and structuring of rhythm and of 12-note equal temperament with major and minor modes.

15.3.1 Beat Tracking

Computational beat tracking is the use of a computer to extract the metrical structure of music (actually a culturally consensual and cognitive construction). In certain musicologically unambiguous situations, this can correspond to the location of beats where average listeners would clap their hands or tap their feet to a musical signal. In more complicated settings, the extraction of meter is accomplished by identifying multiple metrical levels, any duple/triple hierarchical structure, or even other metrical frameworks outside the canon of conventional Western music theory; phase and period (as well as further marking events or patterns) must be determined in more difficult cases. For instance, beat tracking of Balkan dance music (*aksak*) or Norwegian

Hardanger fiddle music would require the resolution of higher-level patterns than any simple isochronous beat, and generalized meter tracking would precede any notion of beat subdivision.

Methods for computational beat tracking vary from rule-based systems for symbolic data to correlation and oscillator methodologies for audio feature data (Gouyon and Dixon, 2005; Collins, 2006). In engineering practice, the beat-tracking model that infers the current metrical state may be distinct from an observation front end that collates evidence from the audio stream.

That's enough theory. Figure 15.5 presents a practical beat tracker for you to try.

```
b = Buffer.alloc(s,1024,1); //for sampling rates 44100 and 48000
//b = Buffer.alloc(s,2048,1); //for sampling rates 88200 and 96000

//track audio in (try clapping a beat or beatboxing, but allow // u
p to 6 seconds for tracking to begin); events will be // spawned at
quarter, eighth and sixteenth note rates
(
x= SynthDef(\help_beattrack2,{
   var trackb,trackh,trackq,tempo;
   var source;
   var bsound,hsound,qsound;
   source = SoundIn.ar(0);
   #trackb,trackh,trackq,tempo = BeatTrack.kr(FFT(b.bufnum,   sourc
e));
   bsound = Pan2.ar(LPF.ar(WhiteNoise.ar*(Decay.kr(trackb,   0.0
5)),1000),0.0);
   hsound = Pan2.ar(BPF.ar(WhiteNoise.ar*(Decay.kr(trackh,0.05)),
3000,0.66),-0.5);
   qsound = Pan2.ar(HPF.ar(WhiteNoise.ar*(Decay.kr(trackq,0.05)),
5000),0.5);
   Out.ar(0, bsound+hsound+qsound);
}).play;
)
x.free;
b.free; // Free the buffer
```

Figure 15.5
BeatTrack.

There are weaknesses in tracking that should be acknowledged, some of them particular to the `BeatTrack` UGen and some also in common with the current state of the art. `BeatTrack` uses a window of 6 s to obtain a stable estimate of the current tempo

and beat phase, but this assumes that the tempo and metrical state do not change during that time. You will find that if you stick to a particular reasonable midtempo periodic pattern, `BeatTrack` will synchronize, but you can easily lose the beat tracker by sudden shifts of tempo and continuous slowing down or speeding up. The trade-off of stability of estimate and reactivity to change is fundamental in such work.

The `BeatTrack2` UGen in the standard distribution provides an alternative algorithm in which the trade-off of the reactivity and stability of the estimate can be directly controlled by choosing the size of the temporal window, among other arguments. `BeatTrack2` makes a majority decision on the winning period and phase hypothesis by analyzing a number of user-specifiable feature stream inputs in parallel. (The Help file gives more details and examples.)

The readers might be curious whether the beat so extracted can drive algorithmically composed parts. An example of how this can be done server-side is implicit in the beat-tracking example, but setting up synchronization via the language is harder. This is the research motive behind my `BBCut2` extensions to SuperCollider. I won't go into them here; instead, I refer the curious reader to that code library and the associated doctoral thesis of Collins (2006). The readers might also explore examples of synchronization by Florian Paul Schmidt (`OSCClocks`, available as a Quark), Fredrik Olofsson (sync by successive tempo adjustments), and James Harkins (`MIDISyncClock`), among others. But as a first step, modifying the previous example, the quarter-note-level beat can be passed to the language by using `SendTrig` to drive the scheduling of actions language-side.

Beat tracking raises an interesting point about machine listening; many of the processes we explore are based around reaction to a signal that is as fast as possible. This is not how a human musician works, though: humming along to a previously unknown melody at a latency of less than 20 ms is impossible for a human but plausible for a computer. Instead, expectation and anticipation are key to human music making, and we are adept at synchronizing ourselves within a performance context, such that the actions that are scheduled in advance will synchronize so long as our predictions are not astray.

15.3.2 Key Tracking

As a further demonstration of higher-level machine listening, an audio signal can be analyzed for the current key signature (Gómez, 2006). This process assumes that 12-tone equal temperament and major/minor tonality make sense within the musical context being examined, and the particular UGen here described, `KeyTrack`, also assumes a tuning of concert A = 440 Hz. To give a short rather than a long explanation, analysis proceeds by matching certain harmonic templates to the spectrum and taking the best-fitting one as indicating the key. Figure 15.6 gives example code.

```
//straight forward test file with few transients; training set // i
n e minor from MIREX2006
//You will need to substitute your own soundfile to load here
d=Buffer.read(s,"/Users/nickcollins/Desktop/ML/training_wav/78.wav"
b = Buffer.alloc(s,4096,1); //for sampling rates 44100 and 48000
//b = Buffer.alloc(s,8192,1); //for sampling rates 88200 and 96000

(
x= {
   var in, fft;
   var key;
   in = PlayBuf.ar(1,d.bufnum,BufRateScale.kr(d.bufnum),1,0,1);
   fft = FFT(b.bufnum, in);
   key = KeyTrack.kr(fft, 2.0, 0.5);
   key.poll; //write out detected key
   Out.ar(0,Pan2.ar(in));
}.play
)

x.free;
b.free;
```

Figure 15.6
KeyTrack.

KeyTrack can be thrown by transient-rich signals—where the audio files to be tracked contain a lot of percussive nonpitched instruments, tracking is less reliable—but the process is designed to operate on polyphonic audio to start with, so don't be afraid to test it on Blondie or Beethoven.

15.3.3 Event Transcription

It is possible to combine pitch detection and onset detection to attempt to transcribe a melodic line over time. We can analyze a sequence of musical events, storing the individually extracted "notes" in a database for algorithmic reuse.

Figure 15.7 presents a relatively minimal example to demonstrate this process, combining the `Pitch` and `Onsets` UGens and storing the last *n* notes (*n* = 10 by default). The quality of transcription is at the mercy of the onset detection; new onsets are the cue to resolve the previous note event and add it to the database. Why does this process resolve the previous note event rather than the current one? The reason is that a new onset notification appears at the start of a note, and pitch will not necessarily have settled yet in the transient region to allow an accurate reading. The code shows how to

record a succession of frequency values language-side and take the median for a stable pitch estimate. The code again demonstrates that machine listening must often operate at a delay.

```
(
var freqlist=List(), amplist=List();
var notelist= List(), numnotes=10; //will hold the last 10 notes
var lasttime, started=false;
var maxlength=0.5, maxkperiods, waittime;
var freqbus = Bus('control',0,1);
var hasfreqbus = Bus('control',1,1);
var rmsampbus = Bus('control',2,1);

maxkperiods = ((maxlength*(s.sampleRate))/(s.options.blockSize)).
asInteger;
waittime = (s.options.blockSize)/(s.sampleRate);

// register to receive message
a = OSCFunc({arg msg, time, addr, recvPort;
   var newnote;
   if(started,{

   //finalise previous note as [starttime, ioi= inter onset // inte
rval, dur, medianpitch, maxamp]
   newnote =   [lasttime, time-lasttime, (time-lasttime).   min(max
length), if(freqlist.notEmpty, {freqlist.median.   cpsmidi},{nil}),
amplist.maxItem.ampdb];

   newnote.postln;

   notelist.addFirst(newnote);

   //remove oldest note if over size
   if(notelist.size>numnotes,{notelist.pop});

   },{started = true;});

   //reset lists for collection
   freqlist = List();
   amplist = List();
   lasttime = time;
```

```
},'/tr',s.addr).add;
x = Synth(\pitchandonsets);

//poll values
{

   inf.do{
        var freq, hasfreq, rmsamp;

        freq = freqbus.getSynchronous;
        hasfreq = hasfreqbus.getSynchronous;
        rmsamp = rmsampbus.getSynchronous;

        //don't allow notes of longer than 500 control periods // o
r so
        if((hasfreq>0.5) and: (amplist.size<maxkperiods),   {freqli
st.add(freq)});

        if(amplist.size<maxkperiods, {amplist.add(rmsamp)});

        //poll every control period, intensive
        (waittime).wait;
   };

}.fork;
)

(
a.free; //Free the OSCFunc
x.free; // Free the synth
b.free; // Free the buffer
)
```

Figure 15.7
Simple melodic transcription.

For a more sophisticated version of this process, the reader might investigate the `AnalyzeEvents` UGens accompanying my thesis work on interactive music systems.

15.4 Interactive Music Systems

15.4.1 Example Systems

The term *interactive music system*, following the title of Robert Rowe's first book (Rowe, 1993), generally covers a panoply of startling, fascinating, and musically provocative creations (Collins, 2007). These typically involve some independence of computer action in response to human input, emulating musical dialogue, though there are many variations of this task.

It's hard to go through a survey of interactive music systems without mentioning George Lewis and Voyager. Lewis's contributions to the field include a great vigor with which he has defended, and indeed promoted, the cause of improvisation as the quintessential musical paradigm; in this vein, some have considered interactive improvisation the most challenging of the test cases for virtual musicianship. Voyager itself is an intentionally chaotic and quixotic system, an openly personal set of rules, a reputed million lines of Forth code now converted to a monster Max/MSP patch. It sounds … well, chaotic and quixotic. It sends out a barrage of MIDI messages—usually, these days, to control the refined acoustic sound source of a Disklavier piano (perhaps what Lewis spent his MacArthur Genius Fund grant money on)—though the now out-of-print 1992 CD involves a general MIDI set of voices. The input can be any monophonic instrument which works with a standard pitch-to-MIDI converter; Lewis enjoys a little extra noise on the line, so perfect conversion isn't the name of the game. This is indeed a pragmatic strategy with the technology. It's all great fun and very inspiring, but not so well documented even in Lewis's academic papers. Although hardly the most high-tech or flexible of solutions, it is extensively gig-tested.

To give a second example, this time of a SuperCollider-based system (originally developed for SuperCollider 2), Joel Ryan has combined an Eventide Harmonizer and additional SuperCollider-based audio processing. He has explored many collaborations with improvising musicians, including the virtuoso of circular breathing, the perpetually-in-motion saxophonist Evan Parker; Ryan is actively involved in the control loop here, and the system is not intended to be autonomous during a concert but a conduit for human musical action.

There is no space to delve further into the rich history and musical activity of these systems, but for audio-based work, the UGens that have been described in this chapter are essential props to effective interactive concert work.

15.4.2 A Language-side Symbolic System Using MIDI

An interactive music system does not necessarily require an audio signal. Robert Rowe's books (Rowe, 1993, 2001) predominantly treat the case of complex MIDI processing systems; and the MIDI representation, though weak in some ways, is a familiar discrete event representation that works especially well for keyboard instruments. In order to illustrate this case, avoiding any audio-based listening process, the code for this chapter includes the `OnlineMIDI` class and associated Help file.

`OnlineMIDI` is a demonstration that should give an insight into how to extract features from symbolic information on the language side. It can be productive to consider purely MIDI-based systems in order to focus on compositional logic and artificial intelligence rather than deal with tricky issues of real-time audio signal processing. Figure 15.8 gives some simple client code for controlling this class, with the specification of a particular response function that utilizes the feature data extracted.

```
//Do this first:
MIDIIn.connectAll(verbose: true)
//now:
m = OnlineMIDI();

m.analyse(3,1.0); //3 seconds window, step size of 1.0 seconds

m.data //poll current data

m.status = true; //prints analysis data as it goes
m.status= false;

//use analysis data to formulate responses
(
SynthDef(\beep2,{arg freq=440,amp=0.1, pan=0.0, dur=0.1;
var source;
source= SinOsc.ar(freq*[1,1.007],0,amp*0.5);
Out.ar(0,Pan2.ar(Mix(source)*Line.kr(1,0,dur,   doneAction:2),pa
n))}).send(s);
)

//to echo each note you play on a MIDI keyboard with a sound; // yo
ur SynthDef must have freq and amp arguments, and deal // with dura
tion and freeing the Synth itself.
(
m.playinput= true;
m.inputsynthdef= \beep2;
)

//set a function that gets called after each window is // analysed,
to schedule events over the next second
(
m.response = {|analysis|
   var number;
   number= analysis.density;
```

```
    //number= max(0,(10-(analysis.density))); //inverting number of
  notes playing
    if(analysis.iois.notEmpty, {
          {
          number.do{
          Synth(\beep2, [\freq, analysis.pitches.choose.midicps,   \a
mp, 0.2*(rrand(analysis.volumemin, analysis.   volumemax))]);
          analysis.iois.choose.wait; //could last longer than the //
  next second, but still fun!
          };

          }.fork;
    });
};
)

m.response = nil; //stop
```

Figure 15.8
OnlineMIDI.

Whereas the objective in audio signal machine listening is to move toward discrete objects more easily treated in SuperCollider language code, MIDI can be a useful tool for prototyping ideas for interaction.

15.4.3 Audio-Based Systems

There are continua on multiple dimensions from intelligent signal processing, such as exhibited by the code in figure 15.1, to more complicated virtual musicians who extract multiple layers of information from an audio signal, as well as deliberate complex responses based on musical modeling and artificial intelligence. Within the scope of this chapter, it is possible only to allude to the existence of larger-scale, interactive music systems.

Various examples of audio-based systems for SuperCollider are available, ranging from work by Herve Provini, Dan Stowell, and Julio d'Escriván to the aforementioned interactive music systems available alongside my doctoral thesis, whose full code is free under the GNU GPL. To briefly mention 2 of the 5, *DrumTrack* involves a duel between a human and a machine drummer in which the degree of contest and synchronicity varies over time, utilizing `BBCut2`; the *Ornamaton* operates in the domain of baroque music, adding improvised embellishments to the playing of a harpsichordist and baroque recorder player via specialized onset detection, event analysis, and key tracking.

15.5 Conclusions

There are wider debates surrounding the technical topics in this chapter concerning the musical and philosophical status of interactive music systems. To give a taste of this, we might ask at what point reaction becomes interaction. Evaluation is also a tricky matter, and it might be independently couched from the perspectives of technologist, audience, and performer, concerning engineering and musicological and human–computer interaction criteria. Our systems operate on various continua ranging from intelligent signal processing as a fancy effect to the autonomous behavior of software agents exhibiting independent musicianship.

In practice, there are no general easy solutions; readers are advised to try different machine listening plug-ins and select those that match their application needs. Concerts are not the only avenue of usage, for machine listening might drive live visuals or provide useful sensor information for installations. Although real-time applications have been emphasized in this chapter, it is often productive to consider their use in non-real-time mode or in the prototyping and preparation of material ahead of a concert. (The SCMIR library mentioned in table 15.1 can assist with this; also see chapter 25 in this volume.) I hope that this chapter has given readers a taste of the potential and excitement of these technologies.

Readers desiring to pursue their own implementations of machine-listening processes will need either to work language-side, with MIDI processing—say, implementing symbolic algorithms (Rowe, 2001)—or, for audio-based work, will need to try plugging together and exploring arguments to existing UGens or writing their own UGen plug-ins (see chapter 29). There is a diverse computer music literature to refer to on these topics. For audio systems, there are a number of other open-source code projects which it may be productive to explore, of which a modest list of current options might include Tartini, Praat, Sonic Visualiser and the Vamp plug-ins format, libXtract, CLAM, marsyas, and essentia.

The tools are there at your disposal and will reward your efforts, so you can attempt to create the fantastical virtual musician that you have always dreamed of playing with! There is no danger of making human beings obsolete; instead, you will contribute to a grand discourse in experimental art which dates to the automata of antiquity.

References

Bogdanov, D., N. Wack, E. Gómez Gutiérrez, et al. 2013. “Essentia: An Audio Analysis Library for Music Information Retrieval.” In *14th Conference of the International Society for Music Information Retrieval (ISMIR)*, Curitiba, Brazil.

Collins, N. 2006. “Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems.” PhD thesis, University of Cambridge.

Collins, N. 2007. “Musical Robots and Listening Machines.” In N. Collins and J. d’Escriván, eds., *The Cambridge Companion to Electronic Music*. Cambridge: Cambridge University Press.

Collins, N. 2011. “SCMIR: A SuperCollider Music Information Retrieval Library.” *Proceedings of the International Computer Music Conference*, Huddersfield.

de Cheveigné, A. 2006. “Multiple F0 Estimation.” In D. Wang and G. J. Brown, eds., *Computational Auditory Scene Analysis: Principles, Algorithms, and Applications*. Hoboken, NJ: Wiley/IEEE Press.

Gómez, E. 2006. “Tonal Description of Music Audio Signals.” PhD thesis, Universitat Pompeu Fabra, Barcelona.

Gouyon, F., and S. Dixon. 2005. “A Review of Automatic Rhythm Description Systems.” *Computer Music Journal*, 29(1): 34–54.

Logan, B. 2000. “Mel Frequency Cepstral Coefficients for Music Modeling.” In *Proceedings of the 2000 International Symposium on Music Information Retrieval* (ISMIR). Amherst: University of Massachusetts at Amherst.

Moore, B. C. J. 2004. *An Introduction to the Psychology of Hearing,* 5th ed. London: Academic Press.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

Roads, C. 2001. *Microsound*. Cambridge, MA: MIT Press.

Rowe, R. 1993. *Interactive Music Systems*. Cambridge, MA: MIT Press.

Rowe, R. 2001. *Machine Musicianship*. Cambridge, MA: MIT Press.

Stowell, D., and M. Plumbley. 2007. “Adaptive Whitening for Improved Real-Time Onset Detection.” In *Proceedings of the 2007 International Computer Music Conference*, Copenhagen.

Tenney, J. 2006. *MetaHodos and META MetaHodos*. Oakland, CA: Frog Peak.

Tremblay, P.A., G. Roma, and O. Green. 2021. “Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit.” *Computer Music Journal*, 45(2): 9–23.

Wishart, T. 1985. *On Sonic Art*. York, UK: Imagineering Press.

Zwicker, E., and H. Fastl. 1993. *Psychoacoustics: Facts and Models,* 2nd ed. Berlin: Springer.

16 Microsound

Alberto de Campo and Marcin Pietruszewski

Since the pioneering work of Dennis Gabor and Iannis Xenakis, the idea of composing music and sound from minute particles has been an interesting domain for both scientific and artistic research. The seminal book on the topic, *Microsound* (Roads, 2001a), covers historical, aesthetic, and technical considerations and provides an extensive taxonomy of variants of particle-based sound synthesis and transformation. Microsound also has an interesting cultural history as an underground genre with a cult following through the microsound.org[1] list and community, which has triggered interesting studies (e.g., Born & Haworth, 2017),

This chapter provides a collection of detailed example implementations of many fundamental concepts of particle-based synthesis, which may serve as starting points for personal adaptations, extensions, and further explorations by readers.

16.1 Points of Departure

Imagine a sine wave that began before the big bang and will continue until past the end of time. If that is difficult, imagine a pulse that has infinite amplitude but also is infinitely short, so its integral is precisely 1.

In 1947 Dennis Gabor answered the question “What do we hear?” in an unusual way: instead of illustrating quantum wave mechanics with acoustical phenomena, he did the opposite: applying a formalism from quantum physics and information theory to auditory perception, he obtained an *uncertainty relation* for sound. By treating signal representation (which is ignorant of frequency) and Fourier representation (which knows nothing about time) as the extreme cases of a general particle-based view on acoustics, he introduced acoustic quanta of information that represent the entity of maximum attainable certainty (or minimum uncertainty). He posited that sound can be decomposed into elementary particles, which are vibrations with stationary frequencies modulated by a probability pulse; in essence, this is an envelope shaped like a Gaussian distribution function. Abraham Moles (1966) emphasized the quantum nature of auditory perception and application of information theory to the arts more generally. These views influenced Iannis Xenakis by 1960 (Xenakis, 1992) to consider sounds as masses of particles that he called “grains” that can be shaped by mathematical means.

Sounds at the micro time scale (below about 100 milliseconds) became accessible for creative experiments following the general availability of computers. A number of pioneers have created programs to generate sound involving decisions at the micro and sample time scales. These are often described as "nonstandard synthesis" (see e.g., Döbereiner 2011) and include Herbert Brün's SAWDUST (1976), G. M. Koenig's SSP (early 1970s), and Iannis Xenakis's Stochastic Synthesis (described first in Xenakis (1971) and realized as the GENDY program in 1991); see also Hoffmann (2000), Luque (2006), and Collins's Gendy UGens for SuperCollider).

Following concepts by Xenakis, Curtis Roads began experimenting with granular synthesis on mainframe computers in 1974; and Barry Truax implemented the earliest real-time granular synthesis engine on special hardware beginning in 1986 and realized the pieces *Riverrun* and *Wings of Nike* with this system (Truax, 1988). Trevor Wishart (1994) has called for a change of metaphor for music composition based on his experience of making electroacoustic music: rather than architecture (of pitch/time/parameter constructions), chemistry or alchemy can provide models for the infinite malleability of sound materials in computer music. This extends to the micro time scale, where it is technically possible to obtain nearly infinite differentiation in creating synthetic grains, though this is perceptually constrained by limitations of differentiation in human hearing. Horacio Vaggione has explored the implications of musical objects at different time scales both in publications (Vaggione, 1996, 2001) and in fascinating pieces (*Agon*, *Nodal*, and others).

Many physical sounds can be described as granular structures: dolphins communicate by clicking sounds; many insects produce micro-sounds; e.g., crickets make friction sounds with a rasping action known as "stridulation," filtered by mechanical resonance of parts of their exoskeletons); bats echolocate obstacles and possible prey by emitting short ultrasound bursts and listening to the returning sound reflections. Many sounds that involve a multitude of similar objects interacting will induce global percepts with statistical properties: rustling leaves produce myriad single short sounds, as do pebbles when waves recede from the shore; the film sound staple of steps on gravel can be perceived in these terms, as can bubbles in liquids, whether they are in a brook or in a frying pan.

Some musical instruments can be described and modeled as granular impact sounds passing through filters and resonators: güiro, rainstick, rattles, maracas, fast tone repetitions on many instruments, fluttertongue effects on wind instruments, and any instrument that can be played with drum rolls.

The microsound perspective also can be applied fruitfully in sound analysis, as initiated by Gabor. Wavelet analysis can decompose signals into elementary waveforms (Kronland-Martinet, 1988), and more recently Wavelet Analysis and Dictionary-Based methods have been employed to create granular representations of sound which offer

new possibilities for sound visualization and transformation (Sturm et al., 2006, 2008, 2009).

Concatenative Synthesis (e.g., Schwarz 2005) also works well at microsound timescales: analysis-informed segmentation of large corpora of recorded sound provides databases which can be used to assemble sequences of short segments by user-given criteria. FluCoMa[2] (short for Fluid Corpus Manipulation) is an interesting recent project that combines a large variety of sound analyis/decomposition and machine learning tools and provides support interfaces for MaxMSP, Pure Data, and SuperCollider to enable musicians to explore these possibilities creatively (Tremblay, Roma, & Green, 2021).

16.2 Perception at the Micro Time Scale

As human listeners perceive sound events differently depending on the time scales at which they occur, it is informative to experiment with the particularities of perception at the micro time scale. While the literature on psychoacoustics (Moore, 2004; Buser and Imbert, 1992) and temporal processing of sound (Snyder, 2000) is rich and interesting, the experiments tend to deploy a rather limited repertoire of test sounds. For exploring new sound material, making one's own experiments (informed by psychoacoustics) can provide invaluable listening experiences.

Pulses repeating at less than about 16 Hz will appear to most listeners to be individual pulses, while pulses at 30 Hz fuse into continuous tones. A perceptual transition happens in between:

```
{Impulse.ar(XLine.kr(12, 48, 6)) * 0.1 ! 2}.play; // up
{Impulse.ar(XLine.kr(48, 12, 6)) * 0.1 ! 2}.play; // down
{Impulse.ar(MouseX.kr(12, 48, 1)) * 0.1 ! 2}.play; // by cursor
```

We are very sensitive to periodicities at different time scales; periodic pulses are perceived as pitches if they are repeated often enough, but how often is enough? With very short-duration tones (on the order of 10 waveform repetitions), one can study how a pitched tone becomes more like a click with different timbre shadings (see figure 16.1).

```
(
// a Gabor grain approximation with a sine-shaped envelope
SynthDef(\gabor1, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
   var snd = FSinOsc.ar(freq);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
```

```
}, \ir ! 5).add;
)

(
Pbindef(\grain,
  \instrument, \gabor1, \freq, 1000, \dur, 1,
  \sustain, Pseq([0.001, 0.1], inf),
  \amp, Pseq([0.1, 0.1], inf)
).play
);
    // short grain 2x louder
Pbindef(\grain, \sustain, Pseq([0.001, 0.1], inf),\amp,   Pseq([0.
2, 0.1], inf));
    // short grain 4x louder
Pbindef(\grain, \sustain, Pseq([0.001, 0.1], inf),\amp,   Pseq([0.
4, 0.1], inf));
```

Figure 16.1
Short grain durations—pitch to colored click.

Very short grains seem softer than longer ones because loudness impressions are formed over larger time windows (in psychoacoustics, this is called "temporal integration"). One can try comparing two alternating grains and adjusting their amplitudes until they seem equal.

With nearly rectangular envelope grains, the effect is even stronger. (See an example in the book code repository.) Short silences imposed on continuous sounds have different effects depending on the input sound: on steady tones, short pauses seem like dark pulses; only longer ones seem like silences; and short interruptions on noisier signals may be inaudible (see figure 16.2).

```
(
p = ProxySpace.push;

~source = {SinOsc.ar * 0.1};
~silence = {|silDur=0.01|
   EnvGen.ar(
        Env([0, 1, 1, 0, 0, 1, 1, 0], [0.01, 2, 0.001, silDur,   0.
001, 2, 0.01]),
        doneAction: 2) ! 2
};
~listen = ~source * ~silence;
~listen.play;
)
```

```
~silence.spawn([\silDur, 0.001]); // sounds like an added pulse
~silence.spawn([\silDur, 0.003]);
~silence.spawn([\silDur, 0.01]);
~silence.spawn([\silDur, 0.03]);        // a pause in the sound

   // try the same examples with noise:
~source = {WhiteNoise.ar * 0.1};
```

Figure 16.2
Perception of short silences.

When granular sounds are played almost simultaneously, the order in which they occur becomes difficult to discern. Figure 16.3 plays two sounds with an adjustable lag between them. Lags above around 0.03 second are easily audible; shorter ones become more difficult.

```
// As grains move close together, their order becomes hard to perce
ive
(
   // a simple percussive envelope
SynthDef(\percSin, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var env = EnvGen.ar(
        Env.perc(0.1, 0.9, amp), timeScale: sustain,   doneAction:
 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;
)
(
Pbindef(\lo,
   \instrument, \percSin, \sustain, 0.05,
   \freq, 250, \amp, 0.2, \dur, 0.5, \lag, 0
).play;
Pbindef(\hi,
   \instrument, \percSin, \sustain, 0.05,
   \freq, 875, \amp, 0.1, \dur, 0.5, \lag, 0
).play;
)
// try different lag times:
Pbindef(\hi, \lag, 0.05); // hi is clearly late
Pbindef(\hi, \lag, 0.03); // still late
Pbindef(\hi, \lag, 0.015); // late or together?

// shift hi by a random time—can you hear which plays first?
```

```
Pbindef(\hi, \lag, ([−1, 1].choose * 0.015).postln);
// is the order easier to hear when the sounds are panned apart?
Pbindef(\hi, \pan, 0.5); Pbindef(\lo, \pan, −0.5);
Pbindef(\hi, \pan, 0); Pbindef(\lo, \pan, 0);
```

Figure 16.3
Order confusion with sounds in fast succession.

In fast sequences of granular sounds, the order is hard to discern, as the grains fuse into a single sound object. But when the order changes, the new composite sounds different (see figure 16.4).

```
   // when their order changes, the sound is subtly different.

(
Pbindef(\grain4,
   \instrument, \percSin, \sustain, 0.03, \amp, 0.2,
   \freq, Pshuf([1000, 600, 350, 250]), // random sequence   every
 each time
   \dur, 0.015
).play;
   // repeat grain cluster
Tdef(\grain, {loop {Pbindef(\grain4).play; 1.wait}}) .play;
)
   // fixed order every time—audible?
Pbindef(\grain4, \freq, Pseq([1000, 600, 350, 250].scramble));
   // back to different order every time
Pbindef(\grain4, \freq, Pshuf([1000, 600, 350, 250]));
```

Figure 16.4
Multiple grains fuse into one composite.

Such small experiments to learn how we hear sounds with unusual properties are helpful for developing perceptually intriguing and effective sound worlds and may be fruitful starting points for systematic studies.

16.3 Grains and Clouds

Any sound particle shorter than about 100 ms (but this is not a hard limit, only an order of magnitude) can be considered a grain, and can be used for creating groups of sound particles. Such groups may be called “streams” or “trains” (if they comprise regular sequences) or “clouds” (if they are more varied). We will first look at the details of

single grains, and then at the properties that arise as groups of them form streams or clouds.

16.3.1 Grain Anatomy

A “grain” is a short sound event consisting of a signal or waveform and an envelope. The waveform can be generated synthetically, taken from a fixed waveform, or selected from recorded material and possibly processed. The envelope is an amplitude shape imposed on the waveform and can strongly influence the grain’s sound character. We restrict our initial examples to simple synthetic waveforms and experiment with the effects of different envelopes, waveforms, and durations on single grains. We can create an envelope and a waveform signal as arrays; in order to impose the envelope shape on the waveform, they are multiplied, and the three signals are plotted, as shown in figure 16.5.

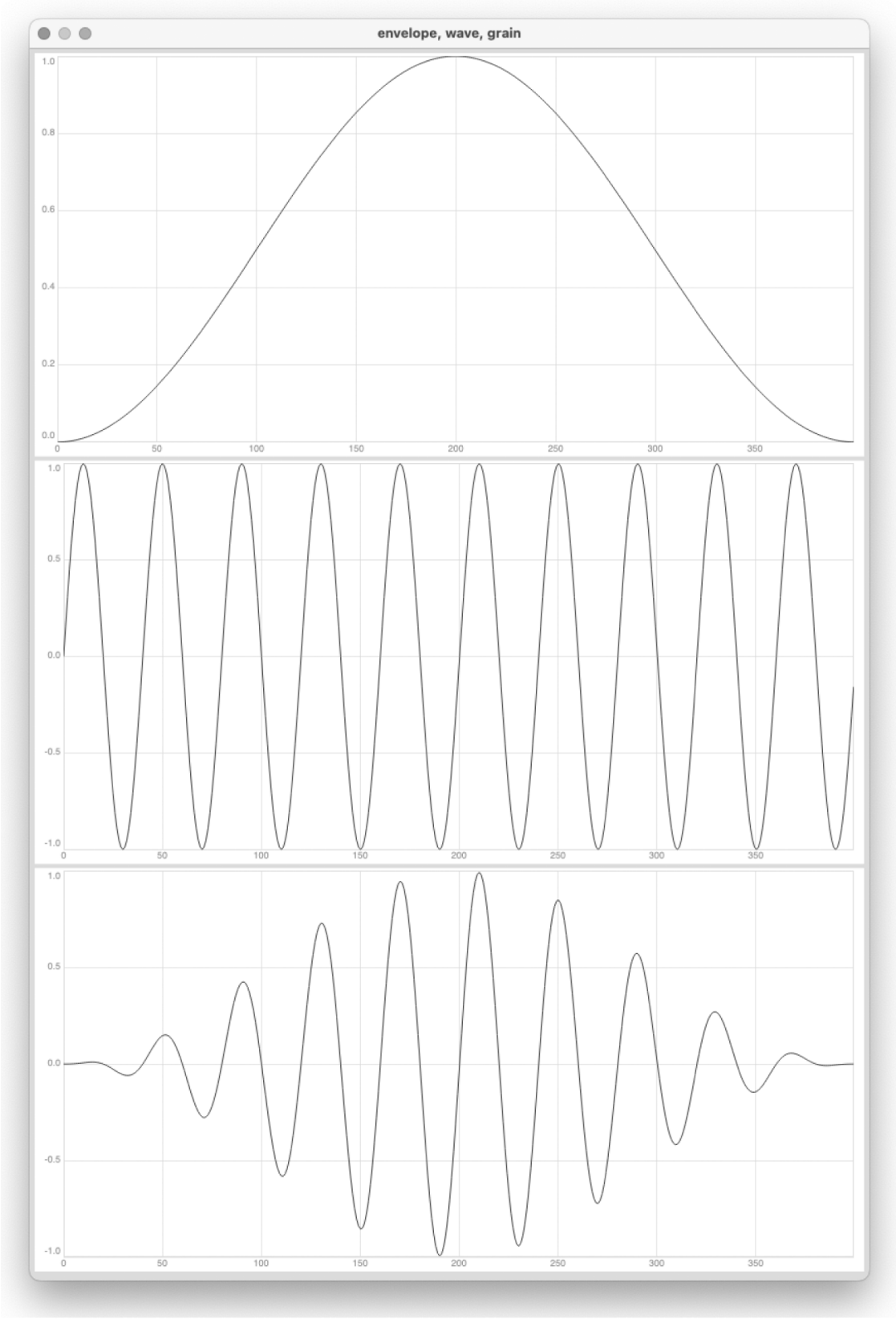


Figure 16.5
Plot of envelope, waveform, and grain.

```
(
var e, w, g;
e = Env.sine.asSignal(400).as(Array);
w = Array.fill(400, {|i| (i * 2pi/40).sin});
g = e * w;
[e, w, g].flop.flat.plot("envelope, wave, grain", Rect(0, 0,   408,
600), numChannels: 3);
)
```

`Env.sine` is close to a Gaussian envelope, providing reasonable approximations of sound quanta as postulated by Gabor. The `LFGauss` UGen can be used as a higher-precision Gaussian envelope for grains, see `{LFGauss.ar(0.01, 0.26)} .plot` or the `LFGauss` Help file.

Assembling these elements in a SynthDef allows for creating many variants of one kind of grain by varying waveform, frequency, grain duration, amplitude, and spatial position:

```
(
SynthDef(\gabor0, {|out, freq = 440, sustain = 0.02, amp = 0.2,   p
an|
   var env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
   var sound = SinOsc.ar(freq) * env;
   OffsetOut.ar(out, Pan2.ar(sound, pan))
}, \ir ! 5).add;
)
Synth(\gabor0);   // test with synth
Synth(\gabor0, [\freq, 1000, \sustain, 0.005, \amp, 0.1,   \pan, 0.
5]);
(instrument: \gabor0).play; // test with event
(instrument: \gabor0, sustain: 0.001, freq: 2500, amp: 0.5,   pan:-
0.5). play;
// higher efficiency, as no Synth object made
Synth.grain(\gabor0, [\freq, 2000, \sustain, 0.003])

// even more efficient as direct message to server.
s.sendMsg("s_new", \gabor0, −1, 0, 0, \freq, 2000, \sustain,   0.00
3);
```

This example demonstrates recommended practices for granular synthesis: as grains may be extremely short, audio-rate envelopes are necessary for precision. Timing between grains should be as accurate as possible, so we use `OffsetOut` to start the grain with single-sample accurate timing. Since parameters are typically fixed for each

grain, we can use `\ir` arguments. The final lines show further optimizations: `Synth.grain` creates nodes without `nodeID` and `Synth` objects (reducing overhead), and `s.sendMsg` does the same even more efficiently with `nodeID -1`, but the result is less comfortable to read. Finally, tests for granular `SynthDefs` should check that every parameter is correctly changeable.

SuperCollider allows quick testing of envelope variants; next, we create a number of common envelopes which can have very different effects on the sound character of the grain: *Gaussian* envelopes minimize the spectral side effects of the envelope, leaving much of the waveform character intact; *quasiGaussian* envelopes increase grain energy by holding full amplitude in the middle of its duration (*welch* interpolation is similar); *exponential decay* creates grains which can sound like they come from a physical source because physical resonators decay exponentially; *reverse exponential decay* can be an intriguing special case of "unnaturalness"; and *percussive* envelopes with controllable attack time can articulate different attack characteristics. More complex envelopes (e.g., the *sinc* function) can be created with sampled mathematical functions or taken from recorded material and played with buffers. (See figures 16.6–16.8.)

```
Env.sine.test.plot;  // approx. gaussian
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \sin).test.plot;   // qu
asi-gaussian
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \lin).test.plot;   // 3
 stage line segments.
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \welch).test.plot;   //
 welch curve interpolation
Env([1, 0.001], [0.1], \exp).test.plot;  // exponential decay
Env([0.001, 1], [0.1], \exp).test.plot;   // reverse exponential
 decay
Env.perc(0.01, 0.09).test.plot;
```

Figure 16.6
Making different envelope shapes.

```
(
q = q? ();
q.makeSinc = {|q, num=1, size=400|
   {|x| x = x.linlin(0, size-1, -pi, pi) * num; sin(x) / x} !   siz
e
};
a = q.makeSinc(6);
a.plot(bounds: Rect(0,0,409,200), minval: -1, maxval: 1);
)
```

Figure 16.7
A sinc function envelope as array.

```
(
[  Env.sine,
   Env([0, 1, 1, 0], [0.33, 0.34, 0.33], \sin),
   Env([0, 1, 1, 0], [0.33, 0.34, 0.33], \lin),
   Env([0, 1, 1, 0], [0.33, 0.34, 0.33], \welch),
   Env([1, 0.001], [1], \exp),
   Env([0.001, 1], [1], \exp),
   Env.perc(0.05, 0.95)
]  .collect(_.discretize(400))
        .add(q.makeSinc(6)).clump(4).collect {|fourItems, i|
              fourItems.flop.flat.plot(
                  ["gauss, quasi-gauss, line, welch",
                  "exponential decay, reversed exponential decay,
perc, sinc"] [i],
                  Rect(478 * i + 100, 300, 478, 400),   numChannel
s: 4)
        };
)
```

Figure 16.8
More envelopes plotted.

With `SynthDefs` using these envelopes (see figure 16.9), we can experiment with the fundamental grain parameters: waveform frequency, envelope shape, and grain duration.

```
(   // a gabor (approx. gaussian-shaped) grain
SynthDef(\gabor1, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var amp2 = amp * AmpComp.ir(freq.max(50)) * 0.5;
   var env = EnvGen.ar(Env.sine(sustain, amp2), doneAction: 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;

   // wider, quasi-gaussian envelope, with a hold time in the middl
e.
SynthDef(\gabWide, {|out, amp=0.1, freq=440, sustain=0.01, pan,   w
idth=0.5|
   var holdT = sustain * width;
   var fadeT = 1-width * sustain * 0.5;
   var snd = FSinOsc.ar(freq);
```

```
   var amp2 = amp * AmpComp.ir(freq.max(50)) * 0.5;
   var env = EnvGen.ar(Env([0, 1, 1, 0], [fadeT, holdT, fadeT],
 \sin),
        levelScale: amp2,
        doneAction: 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 6).add;

   // a simple percussive envelope
SynthDef(\percSin, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var amp2 = amp * AmpComp.ir(freq.max(50)) * 0.5;
   var env = EnvGen.ar(
        Env.perc(0.1, 0.9, amp2),
              timeScale: sustain,
              doneAction: 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;
   // a reversed percussive envelope
SynthDef(\percSinRev, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var amp2 = amp * AmpComp.ir(freq.max(50)) * 0.5;
   var env = EnvGen.ar(
        Env.perc(0.9, 0.1, amp2),
              timeScale: sustain,
              doneAction: 2
        );
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;

   // an exponential decay envelope
SynthDef(\expodec, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var amp2 = AmpComp.ir(freq.max(50)) * 0.5 * amp;
   var env = XLine.ar(amp2, amp2 * 0.001, sustain, doneAction: 2);
   OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;

   // a reversed exponential decay envelope
SynthDef(\rexpodec, {|out, amp=0.1, freq=440, sustain=0.01, pan|
   var snd = FSinOsc.ar(freq);
   var amp2 = amp * AmpComp.ir(freq.max(50)) * 0.5;
```

```
    var env = XLine.ar(amp2 * 0.001, amp2, sustain, doneAction: 2)
         * (AmpComp.ir(freq) * 0.5);
    OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;
)
```

Figure 16.9
`SynthDefs` with different envelopes.

Parameter changes can have side effects of interest. For example, the `\rexpodec SynthDef` (reverse exponential decay) envelope ends with a very fast cutoff—when the waveform amplitude is high at that moment, it creates a click transient. A forward `\expodec` may employ this transient for attack shaping: by adjusting the oscillator's initial phase, different attack colors can be articulated. (See the bonus example in the book code repository.)

16.3.2 Textures, Masses, and Clouds

When we shift attention from individual sound particles to textures composed of larger numbers of microsound events, relations between aspects of the individual events in time create a number of emerging perceptual properties. Different terms have been used for these streams: regular sequences are often called *trains*; while *texture* is a rather flexible term used by, among others, Trevor Wishart (1994); Edgard Varèse often spoke of masses and *volumina* of sound; the *cloud* metaphor suggests an interesting vocabulary for imagining clouds of sound particles, inspired by the rich morphologies of clouds in the Earth's atmosphere or in interstellar space, and their evolution in time.

In *synchronous* granular synthesis, the particles occur at regular intervals and form either a regular rhythm at low densities or an emerging fundamental frequency at higher densities. *Quasisynchronous* streams introduce more local deviations, while in *asynchronous* streams, the timing between events is highly irregular; in the latter case, density really becomes a statistical description of the average number of sounding particles per time unit. Figure 16.11 demonstrates some general strategies for controlling cloud parameters, here applied to cloud density:

Fixed values, which create a synchronous stream.

Time-varying values, specified by an envelope pattern, creating accelerando and ritardando.

Random variation within ranges, with density, which creates a transition to asynchronous streams.

Tendency masks, which combine time-varying values with random ranges, such that the range limits change over time.

Parameter-dependent values, a cloud parameter derived from another cloud parameter. The generalization of this approach is called Grainlet Synthesis; see Roads (2001, pp. 125–129).

These strategies can be applied to any control parameter.

```
(
Pbindef(\grain0,
   \instrument, \gabor1, \freq, 500,
   \sustain, 0.01, \dur, 0.2
).play;
)
   // change grain durations
Pbindef(\grain0, \sustain, 0.1);
Pbindef(\grain0, \sustain, 0.03);
Pbindef(\grain0, \sustain, 0.01);
Pbindef(\grain0, \sustain, 0.003);
Pbindef(\grain0, \sustain, 0.001);
Pbindef(\grain0, \sustain, Pn(Pgeom(0.1, 0.9, 60)));
Pbindef(\grain0, \sustain, Pfunc({exprand(0.0003, 0.03)}));
Pbindef(\grain0, \sustain, 0.03);

   // change grain waveform (sine) frequency
Pbindef(\grain0, \freq, 300);
Pbindef(\grain0, \freq, 1000);
Pbindef(\grain0, \freq, 3000);
Pbindef(\grain0, \freq, Pn(Pgeom(300, 1.125, 32)));
Pbindef(\grain0, \freq, Pfunc({exprand(300, 3000)}));
Pbindef(\grain0, \freq, 1000);
   // change synthdef for different envelopes
Pbindef(\grain0, \instrument, \gabor1);
Pbindef(\grain0, \instrument, \gabWide);
Pbindef(\grain0, \instrument, \percSin);
Pbindef(\grain0, \instrument, \percSinRev);
Pbindef(\grain0, \instrument, \expodec);
Pbindef(\grain0, \instrument, \rexpodec);
Pbindef(\grain0, \instrument, Prand([\gabWide, \percSin,   \percSin
Rev], inf));
```

Figure 16.10
Changing grain duration, frequency, and envelope.

Figure 16.11 shows some control strategies applied to density, opening a spectrum of possibilities between synchronous and asynchronous granular streams.

```
(   // synchronous—regular time intervals
Pbindef(\grain0).clear;
Pbindef(\grain0).play;
Pbindef(\grain0,
   \instrument, \expodec,
   \freq, Pn(Penv([200, 1200], [10], \exp), inf),
   \dur, 0.1, \sustain, 0.06
);
)
   // different fixed values
Pbindef(\grain0, \dur, 0.06) // rhythm
Pbindef(\grain0, \dur, 0.035)
Pbindef(\grain0, \dur, 0.02) // fundamental frequency 50 Hz

   // time-varying values: accelerando/ritardando
Pbindef(\grain0, \dur, Pn(Penv([0.1, 0.02], [4], \exp), inf));
Pbindef(\grain0, \dur, Pn(Penv([0.1, 0.02, 0.06, 0.01].scramble,
 [3, 2, 1], \exp), inf));

   // repeating values: rhythms or tones
Pbindef(\grain0, \dur, Pstutter(Pwhite(2, 15), Pfunc(  {exprand(0.0
1, 0.3)})));

   // introducing irregularity—quasi-synchronous
Pbindef(\grain0, \dur, 0.03 * Pwhite(0.8, 1.2))
Pbindef(\grain0, \dur, 0.03 * Pbrown(0.6, 1.4, 0.1))   // slower dr
ift
Pbindef(\grain0, \dur, 0.03 * Pwhite(0.2, 1.8))

   // average density constant, vary degree of irregularity
Pbindef(\grain0, \dur, 0.02 * Pfunc({(0.1.linrand * 3) + 0.9}));
Pbindef(\grain0, \dur, 0.02 * Pfunc({(0.3.linrand * 3) + 0.3}));
Pbindef(\grain0, \dur, 0.02 * Pfunc({(1.0.linrand * 3) + 0.0}));
Pbindef(\grain0, \dur, 0.02 * Pfunc({2.45.linrand.squared}));   //
 very irregular

(   // coupling—duration depends on freq parameter
Pbindef(\grain0,
   \freq, Pn(Penv([200, 1200], [10], \exp), inf),
   \dur, Pfunc({|ev| 20 / ev.freq})
);
)

   // different frequency movement, different timing
```

```
Pbindef(\grain0, \freq, Pbrown(48.0, 96.0, 12.0).midicps);

(   // duration depends on frequency, with some variation—tendency
 mask
Pbindef(\grain0,
   \freq, Pn(Penv([200, 1200], [10], \exp), inf),
   \dur, Pfunc({|ev| 20 / ev.freq * rrand(0.5, 1.5)})
);
)
```

Figure 16.11
Different control strategies applied to density.

Figure 16.12 shows a number of combinations of the control strategies and grain and cloud parameters described so far.

```
(
Pbindef(\grain0).clear;
Pbindef(\grain0,
   \instrument, \expodec,
   \freq, 200,
   \sustain, 0.05, \dur, 0.07
).play;
)
   // time-varying frequency with envelope pattern
Pbindef(\grain0, \freq, Pn(Penv([200, 1200], [10], \exp), inf));
   // random frequency
Pbindef(\grain0, \freq, 400 * Pwhite(−24.0, 24).midiratio);
   // random variation over time
Pbindef(\grain0, \freq, Pn(Penv([400, 2400], [10], \exp), inf)   *
 Pwhite(−24.0, 24).midiratio);

   // panning
Pbindef(\grain0, \pan, Pwhite(−0.8, 0.8)); // random
Pbindef(\grain0, \pan, Pn(Penv([−1, 1], [2]), inf)); // tendency
Pbindef(\grain0, \pan, Pfunc({|ev| ev.freq.explin(50, 5000, −1,
 1)})); // coupled to freq

   // time scattering variants
Pbindef(\grain0, \dur, 0.1 * Pwhite(0.5, 1.5));   // random range
Pbindef(\grain0, \dur, 0.05 * Prand([0, 1, 1, 2, 4], inf));   // rh
ythmic random

   // randomized amplitude
```

```
Pbindef(\grain0, \amp, Pwhite(0.01, 0.2));   // linear
Pbindef(\grain0, \amp, Pwhite(−50, −14).dbamp); // exponential−   m
ore depth
Pbindef(\grain0, \dur, 0.025 * Prand([0, 1, 1, 2, 4], inf));   // c
ould be denser now
   // random amplitude envelopes with Pseg
(
Pbindef(\grain0,
   \amp, Pseg(
   Pxrand([−50, −20, −30, −40] + 10, inf), // level pattern
   Pxrand([0.5, 1, 2, 3], inf), // time pattern
   Prand([\step, \lin], inf) // curve pattern
   ).dbamp
);
)
   // grain sustain time coupled to frequency
Pbindef(\grain0, \sustain, Pkey(\freq).reciprocal * 20).play;
```

Figure 16.12
Control strategies applied to different parameters.

16.3.3 CloudGenMini

CloudGenMini is a reimplementation of CloudGenerator, a classic microsound program written by Curtis Roads and John Alexander, which creates clouds of sound particles based on user specifications. (For the code, see CloudGenMini.scd in the book code repository.) CloudGenMini provides a collection of `SynthDefs` for different sound particle flavors, a `Tdef` that creates the grain cloud based on current parameter settings, cross-fading between stored settings, and a GUI for playing it. This allows for creating clouds based on tendency masks, which was a central feature of CloudGenerator.

The `Tdef(\cloud0)` plays a loop which randomly chooses values for the next grain within the ranges for each parameter given in the current settings. Especially for high-density clouds, using `s.sendBundle` is more efficient than using `Event.play` or patterns.

Synthesis processes with multiple control parameters have large possibility spaces. When exploring them by making manual changes, one may spend much time in relatively uninteresting areas. One common heuristic for finding more interesting zones is to create random ranges. CloudGenMini can create random ranges for all synthesis and cloud parameters within global maximum settings, and can switch or interpolate between eight stored range settings.

CloudGenerator allowed specifying a total cloud duration, start and end values for high and low band limits (the minimum and maximum frequencies of the grain

waveform); and grain duration, density, and amplitude to define a cloud's evolution in time. CloudGenMini generalizes this approach: by providing ranges for all parameters, and by cross-fading between them, every parameter can mutate from deterministic to random variation within a range (i.e., with a tendency mask). For example, a `densityRange` fading from `[10, 10]` to `[1, 100]` creates a synchronous cloud that evolves to asynchronicity over its duration. (CloudGenerator also offered sound file granulation; CloudGenMini leaves this as an exercise for the reader.)

16.4 Granular Synthesis on the Server

The examples so far have created every sound particle as a single synthesis process. SuperCollider also has a selection of UGens that implement granular synthesis entirely on the server. Though one can obtain similar sounds using either approach, experimenting with UGens imposing parameter control is worthwhile, as it may lead to different ideas.

The first granular synthesis UGen in SC was `TGrains`, which granulates sound files (more on this below); with SC version 3.1, `GrainSin`, `GrainFM`, `GrainBuf`, `GrainIn`, and `Warp1` were added. Third-party libraries like sc3-plugins and Quarks are worth checking for more granular synthesis variants.

`GrainSin` creates a stream of grains with a sine waveform and a Hanning-shaped envelope; grains are triggered by a control signal's positive zero-crossing. Here is the `GrainSin` Help file example converted to JITLib style (see chapter 7):

```
p = ProxySpace.push;
(
~grain.play;
~grain = {|envbuf = -1, density = 10, graindur = 0.1, amp = 0.2|
   var trig = Impulse.kr(density);
   var pan = MouseX.kr(-1, 1); // use horizontal cursor for   panni
ng
   // use WhiteNoise and vertical cursor for deviation from center
   var freqdev = WhiteNoise.kr(MouseY.kr(400, 0));
        GrainSin.ar(2, trig, graindur, 440 + freqdev, pan,   envbu
f) * amp
};
)
```

`GrainSin` can use custom grain envelopes given as buffers. In the next example, an `Env` object is first converted to a `Signal` and then to `abuffer`, and the `~grain` proxy is set to use that buffer. A `bufnum` of `-1` resets it to the built-in envelope:

```
q = q? (); // make a dictionary to keep things around
   // make an envelope and send it to a buffer
q.percEnv = Env([0, 1, 0], [0.1, 0.9], -4);
q.percBuf = Buffer.sendCollection(s, q.percEnv.discretize, 1);
~grain.set(\envbuf, -1); // switch to built-in envelope
~grain.set(\envbuf, q.percBuf.bufnum); // or customized
```

Besides setting parameter controls to fixed values, one can also map control proxies to them:

```
~grain.set(\density, 20);
~grain.set(\graindur, 0.03);
   // map a control proxy to a parameter
~grdur = 0.1; ~grain.map(\graindur, ~grdur);
~grdur = {LFNoise1.kr(1).range(0.01, 0.1)}; // random graindur
~grdur = {SinOsc.kr(0.3).range(0.01, 0.1)}; // periodic
~grdur = 0.01; // fixed value
   // create random densities from 2 to 2 ** 6, exponentially distr
ibuted
~grdensity = {2 ** LFNoise0.kr(1).range(0, 6)};
   // map to density control
~grain.map(\density, ~grdensity);
```

At this point, exploration becomes more enjoyable with an `NdefGui` on the proxy and adding `Specs` for its parameters with `Spec.add`. (See the `NdefGui` Help file.)

The `GrainFM` UGen introduces a variant: as the name implies, a pair of sine oscillators create frequency-modulated waveforms within each grain. What follows is the `GrainFM` Help file example rewritten as a `NodeProxy`, with `MouseY` controlling modulation range; such rewrites are useful for learning how different controls affect the sound:

```
// still in pushed ProxySpace p from before
~grain.clear;
~grain = {|envbuf = -1, density = 10, graindur = 0.1, modfreq   = 2
00|
   var pan = WhiteNoise.kr;
   var trig = Impulse.kr(density);
   var freqdev = WhiteNoise.kr(MouseY.kr(0, 400));
   var modrange = MouseX.kr(1, 10);
   var moddepth = LFNoise1.kr.range(1, modrange);
   GrainFM.ar(2, trig, graindur, 440 + freqdev, modfreq,   moddept
h, pan, envbuf) * 0.2
```

```
};
~grain.play;
```

For more flexibility in experimentation, one can convert controls of interest to proxies and access them in the main proxy (in this case, `~grain`). Controls can then be changed individually, and old and new control synthesis functions can be cross-faded. Figure 16.13 shows such a rewrite, where all parameters may be changed freely between fixed values and synthesis functions, line by line, in any order.

```
// still in pushed ProxySpace p from before
(
~trig = {|dens=10| Impulse.kr(dens)};
~freq = {MouseX.kr(100, 2000, 1) * LFNoise1.kr(1).range(0.25,   1.7
5)};
~moddepth = {LFNoise1.kr(20).range(1, 10)};
~modfreq = 200;
~graindur = 0.1;
// rewrite ~grain to use control proxies
~grain = {arg envbuf = -1;
   GrainFM.ar(2, ~trig.kr, ~graindur.kr,
   ~freq.kr, ~modfreq.kr, ~moddepth.kr,
   pan: WhiteNoise.kr, envbufnum: envbuf).postln * 0.2
};
~grain.play;
)
   // change control ugens:
~modfreq = {~freq.kr * LFNoise2.kr(1).range(0.5, 2.0)}; // modfreq
 roughly follows freq
~trig = {|dens=10| Dust.kr(dens)};   // random triggering, same   d
ensity
~freq = {LFNoise0.kr(0.3).range(200, 800)};
~moddepth = 3;   // fixed depth
~graindur = {LFNoise0.kr.range(0.01, 0.1)};

   // bonus: blend dust and impulse triggers
~trig = {|dens=20, bal=0.2| Dust.kr(dens * (1-bal)) + Impulse.   kr
(dens * bal)};
~trig.set(\bal, 0.1);
~trig.set(\bal, 0.5);
~trig.set(\bal, 0.9);
~grain.end; p.pop; // cleanup
```

Figure 16.13
`GrainFM` with individual control proxies.

Finally, we look at the `GrainBuf` UGen, which takes its waveform from a buffer on the server. Typically used for sound file granulation, it can potentially produce variety and movement in the sound stream by varying the file read position (i.e., the location in the sound file where you will take the next grain waveform from) and the playback rate. Even simply moving the file read position along the time axis can create interesting articulation of the granular stream. Figure 16.14 rewrites a `GrainBuf` Help file example with separate control proxies and explores a number of different combinations of controls.

```
p = ProxySpace.push;
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01-   44_
1.aiff");
(
~grain.set(\wavebuf, b.bufnum);
~trig = {|dens=10| Impulse.kr(dens)};
~graindur = 0.1;
~filepos = {LFNoise2.kr(0.2).range(0, 1)};
~rate = {LFNoise1.kr(500).range(0.5, 1.5)};
~grain = {arg envbuf = -1, wavebuf = 0;
   GrainBuf.ar(2, ~trig.kr, ~graindur.kr, wavebuf,
   ~rate.kr, ~filepos.kr, 2, WhiteNoise.kr, envbuf).flat * 0.2
};
~grain.play;
)

   // experiment with control proxies
~trig = {|dens=20| Impulse.kr(dens)};
~rate = {LFNoise1.kr(500).range(0.99, 1.01)};
~filepos = {MouseX.kr + LFNoise0.kr(100, 0.03)};
~graindur = 0.05;
~trig = {|dens=50| Dust.kr(dens)};

c = Buffer.sendCollection(s, Env.perc(0.01, 0.99).discretize, 1);
~grain.set(\envbuf, c.bufnum);
~grain.set(\envbuf, -1);

~trig = {|dens=50| Impulse.kr(dens)}; ~graindur = 0.05;
~grain.end; p.pop; // cleanup
```

Figure 16.14
`GrainBuf` and control proxies.

`TGrains` is very similar to `GrainBuf`, but only with a fixed envelope shape. `GrainIn` is also similar, but it granulates an input signal and can use different buffer envelopes. With the built-in multichannel panning (`PanAz`-like) common to this UGen family, `GrainIn` can be used elegantly for spatially scattering (e.g., continually processing) live input.

When comparing UGen-based and individual-grain particle syntheses, experimenting with UGens as controls allows interesting behavior to occur. Even much patternlike behavior can be realized with `Demand` UGens. (See the Demand help file.) While it is possible to create special flavors with server-side granular synthesis, implementing new UGens requires some fluency in C++ (see chapter 29) and is slower than experimenting with rewriting synthesis patches.

16.5 Exploring Granular Synthesis Flavors

The most flexible starting point for creating one's own microsound flavors is considering that grains may contain any waveform. Reviewing the `SynthDefs` given in figure 16.9, all one needs to change is the sound source itself.

Glisson synthesis is based on Iannis Xenakis's use of glissandi (instead of fixed-pitch notes) as building blocks for some of his instrumental music. Introducing a linear sweep from `freq` to `freq2` is sufficient for a minimal demonstration of the concept. Of course, one can experiment freely with different periodic waveforms and envelopes (see figure 16.15).

```
(
SynthDef(\glisson,
    {|out = 0, envbuf, freq=800, freq2 = 1200, sustain=0.001,   amp
=0.2, pan = 0.0|
        var env = Env.linen(0.1, 0.8, 0.1);
        var envgen = EnvGen.ar(env, timeScale: sustain,   doneActio
n: 2);
        var freqenv = XLine.ar(freq, freq2, sustain);
        OffsetOut.ar(out,
              Pan2.ar(SinOsc.ar(freqenv) * envgen, pan, amp)
        )
}, \ir ! 7).add;
)

(
Tdef(\gliss0, {|e|
```

```
    100.do({arg i;
        s.sendBundle(s.latency, ["/s_new", "glisson", -1, 0, 0,
            \freq, i % 10 * 100 + 1000,
            \freq2, i % 13 * -100 + 3000,
            \sustain, 0.05,
            \amp, 0.1
        ]);
        (3 / (i + 10)).wait;
    });
}).play;
)
```

Figure 16.15
Glisson synthesis.

One possibility for organizing glissando structures is magnetization patterns (Roads, 2001, pp. 121–125). See the example in the book code repository.

Pulsar synthesis is named after pulsars, spinning neutron stars discovered by Jocelyn Bell in 1967 that emit electromagnetic pulses in the range of 0.25 Hz to 642 Hz. For sound waves, this range of frequencies crosses the time scale from rhythm to pitch, a central aspect of Pulsar Synthesis. It also connects back to the history of creating electronic sounds with analog impulse generators and filter responses.

The pulse waveform is determined by a fixed waveform, the *pulsaret,* and an envelope waveform, both of which are scaled to the pulse's duration. Pulsar synthesis was designed by Curtis Roads in conjunction with a special control model: a set of tables which can be edited by drawing is used for designing both waveforms (for pulsaret and envelope) and a group of control functions for synthesis parameters over a given time; this concept has been expanded in the PulsarGenerator program, written in SC2 in 1999–2000 by Alberto de Campo and Curtis Roads. In the following section, we discuss the basic principles of pulsar synthesis as it was realized in this application.

The two main control parameters in pulsar synthesis are fundamental frequency (`fundfreq`), the rate at which pulses are emitted, and formant frequency (`formfreq`), which determines how fast the pulsaret and envelope are played back—effectively like a formant control. For example, at a `fundfreq` of 20 Hz, twenty pulses are emitted per second; at a `formfreq` of 100 Hz, every pulse is scaled to a 0.01-second duration, so within 0.05 seconds of each pulsar period, the duty cycle where signal is present lasts only 0.01 second. Each pulsar train also has controls for amplitude and spatial trajectory. Figure 16.16 shows the creation of a set of tables that are then sent to buffers.

```
(
q = ();
```

```
q.curr = ();   // make a dict for the set of tables
q.curr.tab = ();

   // random tables for pulsaret and envelope waveforms:
q.curr.tab.env = Env.perc.discretize;
q.curr.tab.pulsaret = Signal.sineFill(1024, {1.0.rand} ! 7);

   // random tables for the control parameters:
q.curr.tab.fund = 200 ** Env({1.0.rand}! 8, {1.0.rand} ! 7,   \si
n).discretize.as(Array);
q.curr.tab.form = 500 ** (0.5 + Env({rrand(0.0, 1.0)} ! 8,   {1.0.r
and}! 7, \sin).discretize.as(Array));
q.curr.tab.amp = 0.2! 1024;
q.curr.tab.pan = Signal.sineFill(1024, {1.0.rand} ! 7);

   // make buffers from all of them:
q.bufs = q.curr.tab.collect({|val, key| Buffer.sendCollection(s,
 val, 1)});
)
   // plot one of them
q.bufs.pulsaret.plot("a pulsaret");
```

Figure 16.16
Pulsar basics—making a set of waveform and control tables.

Figure 16.17 realizes one pulsar train with a `GrainBuf`, initially with fixed parameter values. Changing the parameters one at a time and cross-fading between them demonstrates the effect of movements of formfreq and fundfreq. Finally, replacing the controls with looping tables completes a minimal pulsar synthesis program.

```
p = ProxySpace.push;

(
// requires tables made in 16.16
   // fund, form, amp, pan
~controls = [16, 100, 0.5, 0];
~pulsar1.set(\wavebuf, q.bufs.pulsaret.bufnum);
~pulsar1.set(\envbuf, q.bufs.env.bufnum);

~pulsar1 = {|wavebuf, envbuf = -1|
   var ctls = ~controls.kr;
   var trig = Impulse.ar(ctls[0]);
```

```
    var grdur = ctls[1].reciprocal;
    var rate = ctls[1] * BufDur.kr(wavebuf);

    GrainBuf.ar(2, trig, grdur, wavebuf, rate, 0, 4, ctls[3],   envb
uf);
};
~pulsar1.play;
)

    // crossfade between control settings
~controls.fadeTime = 3;
~controls = [16, 500, 0.5, 0]; // change formfreq
~controls = [50, 500, 0.5, 0]; // change fundfreq
~controls = [16, 100, 0.5, 0]; // change both
~controls = [rrand(12, 100), rrand(100, 1000)];

(  // control parameters from looping tables
~controls = {|looptime = 10|
        var rate = BufDur.kr(q.bufs.pulsaret.bufnum) / looptime;
        A2K.kr(PlayBuf.ar(1, [\fund, \form, \amp, \pan].collect(q
.bufs[_]),
              rate: rate, loop: 1));
};
)
```

Figure 16.17
Pulsars as node proxies using `GrainBuf`.

Figure 16.18 shows how to send different tables to the buffers. One can make graphical drawing interfaces for the tables and send any changes in the tables to the associated buffers.

```
// while ~pulsar1 from above is still playing,
// make new pulsaret tables and send them to the buffer:
q.bufs.pulsaret.sendCollection(Array.linrand(1024, -1.0, 1.0));
 // noise burst
q.bufs.pulsaret.read(Platform.resourceDir +/+ "sounds/a11wlk01.   w
av", 44100 * 1.5); // sample
q.bufs.pulsaret.sendCollection(Pbrown(-1.0, 1.0, 0.2).asStream.   n
extN(1024));

    // make a new random fundfreq table, and send it
```

```
q.curr.tab.fund = 200 ** Env({1.0.rand}!8, {1.0.rand}!7, \sin).   d
iscretize.as(Array);
q.bufs.fund.sendCollection(q.curr.tab.fund);

   // and a new random formfreq table
q.curr.tab.form = 500 ** (0.5 + Env({rrand(0.0, 1.0)}!8,   {1.0.ran
d}!7, \sin).discretize.as(Array));
q.bufs.form.sendCollection(q.curr.tab.form);
```

Figure 16.18
Making new tables and send them to buffers.

Pulsar synthesis also can be implemented with sclang-side control, where one can choose to control parameters from patterns such as envelope segment players or from tables (see the examples in the book code repository). This provides elegant control of finer aspects of pulsar synthesis, such as handling pulsar width modulation (what to do when pulses overlap), extensions to parallel pulse trains, and variants of pulse masking.

Among others, Curtis Roads,[3] Florian Hecker,[4] and Marcin Pietruszewski[5] realized a number of pieces with material generated by pulsar synthesis. Pulsars lend themselves particularly well to being exciter signals for filters or input material for convolution processes (Roads, 2001, pp. 147–154). Some wider aesthetic and technocultural aspects of composition with pulsars have also been an object of musicological studies (Haworth, 2015; Pietruszewski, 2020). The following section introduces the New Pulsar Generator (nuPG) program designed by Marcin Pietruszewski.

16.6 The New Pulsar Generator (nuPG) Program

The New Pulsar Generator (*nuPG*) is a reimplementation and extension of the original PulsarGenerator (2001) program discussed above. It is available as source code and as a SuperCollider stand-alone for MacOS.[6]

The historic PulsarGenerator realized several aspects of advanced pulsar synthesis: three parallel pulsar trains (shared fundamental frequency with independent formant frequency, amplitude, and panning controls), burst and stochastic pulsar masking, and saving and cross-fading between sets of tables. The nuPG program extends this with pulsar masking by sieves, per-pulsar multiparameter modulation, frequency modulation, and just-in-time control possibilities (figure 16.19). Since sieves are generally useful, and especially so in microsound, we discuss pulse masking by sieves in detail here; for full details on nuPG, please see its manual.

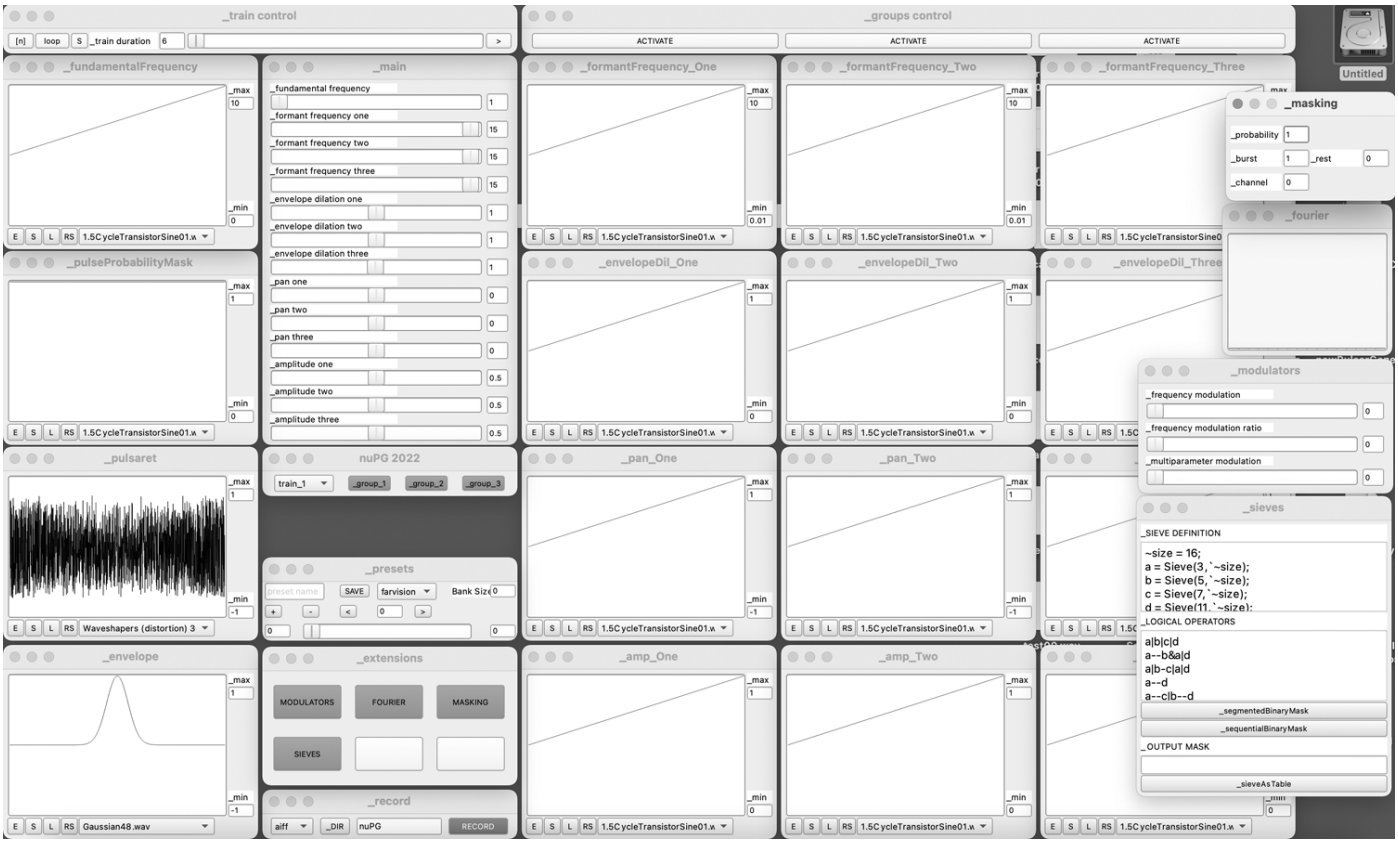


Figure 16.19
A screenshot of the nuPG program.

In microsound synthesis, masking refers to a procedural omission of individual grains from a continuous train. Curtis Roads proposes three techniques of masking for pulsar synthesis (Roads, 2001: 149): *burst, stochastic and channel masking*. The nuPG provides an additional method of masking derived from Iannis Xenakis' sieves (Xenakis and Rahn, 1990).

Burst masking specifies how many pulses to play and how many to mute via a burst-rest ratio. For example, a 3:2 ratio would create this sequence: 1 1 1 0 0, 1 1 1 0 0, and so on. At the infrasonic emission rate, this creates regular rhythmic patterns. At an emission rate in the audio range, burst masking produces a timbral effect akin to amplitude modulation, generating subharmonics of the fundamental frequency.

Stochastic (or *probabilistic*) *pulse masking* uses a probability parameter to randomly mute a certain percentage of pulses. In PulsarGenerator, this probability is expressed as a curve over the whole duration of the pulsar train: when probability equals 1, every pulse is emitted, for 0, all pulses are muted. Values between 0.9 and 0.8 produce an interesting analoglike intermittancy, as if there were an erratic contact in an analog synthesis circuit (Roads, 2001: 151).

Channel masking selectively masks pulsars in two or more channels, creating a dialogue within a phrase by articulating each channel in turn. This can be done by

setting segments of the amplitude table for the respective trains to 0.

Sieve-based masking, by contrast, generates a series of pulsar omission patterns combining sets of integer index numbers through operations of union, intersection, symmetric difference and difference. Xenakis developed sieves as a formal tool to generate subsets of the natural numbers to determine multiple parameters in a composition—pitch sets, time points, loudness or density values, intervals, or local timbres. In short, a sieve can be looked at as a selection of points along the 'line' of natural numbers. The nuPG adaptation builds upon Daniel Meyer's `Sieve` class for SuperCollider (in the `miSCellaneous_lib` Quark) and Christopher Ariza's Python implementation (see Ariza, 2005; Ariza, 2009). Accessible from the GUI shown in figure 16.20), sieve definitions and set operations allow for generating sequences of pulse-omission patterns of varying degrees of complexity.

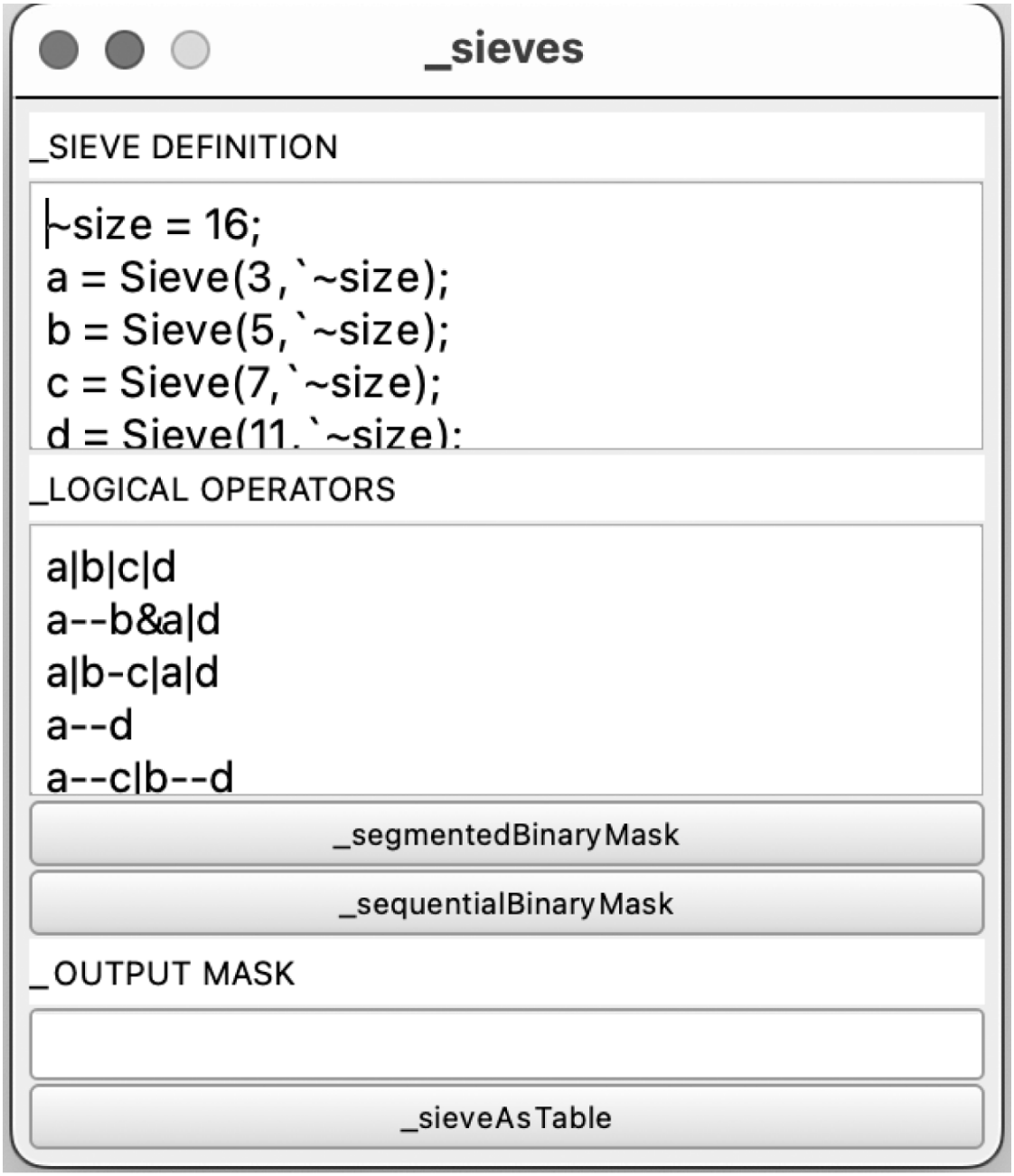


Figure 16.20
The sieve GUI of the nuPG program.

For the purpose of masking, we use two methods for converting sieves to masks. The first method takes the last list element as length and places 1 at every index present in the points list, so a sieve of `[0, 3, 6, 8]` becomes `[1, 0, 0, 1, 0, 0, 1, 0]`. This follows the Python function `discreteBinaryPad` in Ariza's `athenaCL`. We can call this method "sequential sieve binary."

The second method takes the interval notation of a sieve, where the list contains relative steps, not absolute indices; that is, a points list of `[0, 3, 6, 8]` becomes an interval list of `[3, 3, 2]`. Akin to burst masking, each step becomes a series of either 1s or 0s, and these alternate at every step. For example, an input interval list `[3, 3,`

`2]` produces the masking pattern `[0, 0, 0, 1, 1, 1, 0, 0]`. We call this method "segmented sieve binary."

Figure 16.20 shows step-by-step transformations of sieves to masking patterns as used in nuPG. The nuPG sieve GUI supports interactive live coding with sieves for masking, for exploration and performance. The book code repository example also contains simple `Tdefs` that sonify the masking patterns shown.

```
// These use the Sieve class from Daniel Mayer's miSCellaneous_lib
~size = 24; //size limit for all generated sieves

// make 4 sieves to play with
a = Sieve(3, `~size);
b = Sieve(5, `~size);
c = Sieve(7, `~size);
d = Sieve(11 `~size);

//two methods for transforming sieves to pulsar masks:
// sequential bit mask
~pointsToSeqBits = {|points|
   var extractPoints = points.list;
   var bits = 0.dup(extractPoints.last);
   // put 1 at all points indices
   extractPoints.drop(-1).do(bits.put(_, 1));
   bits
};

~pointsToSeqBits.(a); // multiples of 3, as 1 0 0, 1 0 0

// alternating segments bit mask:
~pointsToSegments = {|points|
   points.list.differentiate.collect {|interval, index|
        // at every interval, switch between 1 and 0
        Array.fill(interval, {index.odd.asInteger})
   }.flatten;
};

~pointsToSegments.(a); // multiples of 3, as burst/rest pattern

// set operations on sieves as masking sequences

// union of a and b: all multiples of 3 and 5
~pointsToSeqBits.(a|b);
```

```
// symmetrical difference of 3 and 5—removes common multiples 0 and
15
~pointsToSeqBits.(a—b);

//union of all four—all multiples of 3, 5, 7, 11
~pointsToSeqBits.(a|b|c|d); // as sequence
—> [1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1,   0,
 1, 1, 1, 0]

~pointsToSegments.(a|b|c|d); // as segments
—> [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0,   0,
 1, 0, 1, 1]
```

Figure 16.21
Making, combining, and playing sieves.

The nuPG program also extends the pulsar synthesis model described by Roads with per-pulsar modulation (frequency and multiparameter) and matrix-based parameter modulation. Please see the user manual for details.

16.7 Sound Files and Microsound

Sound files are a good source for waveform material in microsound synthesis, as they provide a lot of variety almost for free, simply by accessing different segments within their duration. Pitch shifting and time scaling are classic uses of granular synthesis in this context. However, further areas to be explored are more complex `SynthDefs` for granulating sound files and processing them per grain (e.g., with filtering) and analyzing sound file waveforms to use them as a source of microstructures. Below we present examples for both: constant-Q granulation and wavesets.

16.7.1 Granular Pitch Shifting and Time Scaling

The idea of being able to manipulate time and pitch separately in a recorded sound is quite old; in the 1940s, Dennis Gabor (who won the Nobel Prize in Physics in 1971 for the invention of holography) built the Kinematical Frequency Convertor, a machine for experimenting with pitch/time manipulations. The principle has remained the same: grains read from a recording are overlapped to create a seamless stream; depending on the location in the recording where the grain read begins and how fast the signal is read, the time and pitch of the original can be changed.

Figure 16.22 provides a setup for experimenting. Here, `pitchRatio` determines how much faster or slower the waveform in each grain is played; `pitchRd` adds randomization to it; `grainRate` is the number of grains per second, and overlap sets how many grains will overlap at any time; `posSpeed` determines whether time is

stretched or compressed by changing the speed of the read position in the file; and `posRd` adds randomization to it. The `NdefGui` interface allows for tweaking the parameters, so, for example, to time-stretch a file by 4, one would set `posSpeed` to 0.25 and adjust `pitchRd` and `posRd` for a compromise between metallic artifacts (common with no randomizing) and scattering artifacts (too much randomness). The `PitchShift` UGen works similarly on input signals.

```
p = ProxySpace.push(s.boot);
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01-   44_
1.aiff");

(
~timepitch = {|pitchRatio=1, pitchRd=0.01, grainRate=10, overlap=2,
   posSpeed=1, posRd=0.01 |
   var graindur = overlap / grainRate;
   var pitchrate = pitchRatio + LFNoise0.kr(grainRate, pitchRd);
   var position = LFSaw.kr(posSpeed / BufDur.kr(b)).range(0, 1)
        + LFNoise0.kr(grainRate, posRd);
   GrainBuf.ar(2, Impulse.kr(grainRate), graindur, b, pitchrate,
               position, 4, 0, -1)
};
~timepitch.play;
);

Spec.add(\pitchRatio, [0.25, 4, \exp]);
Spec.add(\pitchRd, [0, 0.5, \amp]);
Spec.add(\grainRate, [1, 100, \exp]);
Spec.add(\overlap, [0.25, 16, \exp]);
Spec.add(\posSpeed, [-2, 2]);
Spec.add(\posRd, [0, 0.5, \amp]);
NdefGui(~timepitch, 10);
   // reconstruct original
~timepitch.set(\pitchRatio, 1, \pitchRd, 0, \grainRate, 20,   \over
lap, 4, \posSpeed, 1, \posRd, 0);

   // four times as long: tweak pitchRd and posJitter to reduce art
ifacts
~timepitch.set(\pitchRatio, 1, \pitchRd, 0, \grainRate, 20,   \over
lap, 4, \posSpeed, 0.25, \posRd, 0);

   // random read position, random pitch
~timepitch.set(\pitchRatio, 1, \pitchRd, 0.5, \grainRate, 20,   \ov
```

```
erlap, 4, \posSpeed, 0.25, \posRd, 0.5);
~timepitch.end; p.pop; // cleanup
```

Figure 16.22
Time-pitch changing.

Tuning pitch-time changes to sound more lifelike can be difficult, and many commercial applications spend much effort on it. Bringing forward details in recordings in nonnaturalistic ways may be a more rewarding use for writing one's own pitch-time manipulating instruments.

16.7.2 Constant-Q Granulation

Constant-Q granulation is just one of many possible extensions of file granulation. In it, each grain is bandpass filtered, letting the filter ring while it decays. Figure 16.23 shows the `SynthDef` for it: a grain is read around a center position within the file, ring time and amplitude compensation are estimated for the given frequency and resonance, and a cutoff envelope ends the synthesis after ring time is over. High `rq` values color the grain only a little, and low `rq` values create ringing pitches. Because the resonance factor `q` is constant, lower frequencies will ring longer.

```
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01-   44_
1.aiff");
(
SynthDef(\constQ, {|out, bufnum=0, amp=0.1, pan, centerPos=0.5,   s
ustain=0.1,
   rate=1, freq=400, rq=0.3|

   var ringtime = (2.4 / (freq * rq) * 0.66).min(0.5);   // estimat
ed
   var ampcomp = (rq ** -1) * (400 / freq ** 0.5);
   var envSig = EnvGen.ar(Env([0, amp, 0], [0.5, 0.5] * sustain,
 \welch));
   var cutoffEnv = EnvGen.kr(Env([1, 1, 0], [sustain+ringtime,   0.
01]), doneAction: 2);
   var grain = PlayBuf.ar(1, bufnum, rate, 0,
        centerPos-(sustain * rate * 0.5) * BufSampleRate.   ir(bufn
um),
        1) * envSig;
   var filtered = BPF.ar (grain, freq, rq, ampcomp);

   OffsetOut.ar(out, Pan2.ar(filtered, pan, cutoffEnv))
}, \ir ! 8).add;
```

```
)
// test with a Synth first
Synth(\constQ, [\bufnum, b, \freq, exprand(100, 10000), \rq,   expr
and(0.01, 0.1), \sustain, 0.01, \amp, 0.5]);
// Create a stream of constant Q grains
(
Pbindef(\gr1Q,
   \instrument, \constQ, \bufnum, b.bufnum,
   \sustain, 0.01, \amp, 0.2,
   \centerPos, Pn(Penv([1, 2.0], [10], \lin)),
   \dur, Pn(Penv([0.01, 0.09, 0.03].scramble, [0.38, 0.62]   * 10,
 \exp)),
   \rate, Pwhite(0.95, 1.05),
   \freq, Pbrown(64.0, 120, 8.0).midicps,
   \pan, Pwhite(−1, 1, inf),
   \rq, 0.03
).play;
)
   // changing parameters while playing
Pbindef(\gr1Q, \rq, 0.1);
Pbindef(\gr1Q, \rq, 0.01);
Pbindef(\gr1Q, \sustain, 0.03, \amp, 0.08);
Pbindef(\gr1Q, \freq, Pbrown(80, 120, 18.0).midicps);

Pbindef(\gr1Q, \rq, 0.03);

Pbindef(\gr1Q, \rate, Pn(Penv([1, 2.0], [6], \lin)));
   // variable duration
Pbindef(\gr1Q, \dur, Pwhite(0.01, 0.02));

   // a rhythm that ends
Pbindef(\gr1Q, \dur, Pgeom(0.01, 1.1, 40));
```

Figure 16.23
A constant-Q `SynthDef` and `Pattern`.

With this synthesis flavor, one can balance how much of the sound file material shines through and how strongly structures in grain timing and resonating pitches shape the sounds created. Many other transformation processes can be applied to each grain, with individual parameters creating finely articulated textures.

16.7.3 Wavesets

Trevor Wishart (1994) introduced the waveset concept in *Audible Design*, implemented it in the Composer Desktop Project (CDP) framework as a transformation tool, and

employed it in several compositions (e.g., *Tongues of Fire,* 1994). Although Wishart treats wavesets mainly as units for transforming sound material, they can in fact also be used to turn a sound file into a large repository of waveform segments to be employed for a variety of synthesis concepts.

A "waveset" is defined as the waveform segment from one nonpositive to positive zero-crossing of a signal to the next, so in a sine wave, it corresponds to the sine wave's full period. In aperiodic signals (such as sound files), the length and shape of waveform segments vary widely. In the current implementation in the `WavesetsEvent` quark, the `Wavesets2` class breaks a sound signal into wavesets, keeping zero-crossings, lengths, amplitudes, and other values of each waveset (see the `Wavesets2` Help file) for direct access, and the `WavesetsEvent` class provides more sophisticated pattern/event support.

Table 16.1 lists most of Wishart's waveset transforms, with short descriptions. Some of these transformations can be demonstrated with just the `Wavesets` class, a single `SynthDef`, and a `Pbindef` pattern.

Table 16.1
Overview of waveset transforms

Transposition	Take every second waveset, play back at half speed.
Reversal	Play every waveset or group time reversed
Inversion	Turn every half waveset inside out.
Omission	Play silence for every *m* out of *n* wavesets (or randomly by percentage).
Shuffling	Switch every two adjacent wavesets or groups.
Distortion	Multiply waveform by a power factor (i.e., exponentiate with constant peak).
Substitution	Replace waveset with any other waveform (e.g., sine, square, other waveset).
Harmonic distortion	Add double-speed and triple-speed wavesets to every waveset; weight and sum them.
Averaging	Scale adjacent wavesets to average length and average their waveforms.
Enveloping	Impose an amplitude envelope on a waveset or group.
Waveset transfer	Combine waveset timing from one source with waveset forms from another.
Interleaving	Take alternating wavesets or groups from two sources.
Time stretching	Repeat every waveset or group *n* times (creates "pitch beads").
Interpolated time stretching	Interpolate between waveforms and durations of adjacent wavesets over n cross-fading repetitions.
Time shrinking	Keep every *n*th waveset or group.

Figure 16.24 creates a waveset from a sound file and displays accessing its internal data and accessing and plotting individual wavesets or groups of wavesets.

```
// using WavesetsEvent quark:
// ~wsev is the WavesetsEvent which does the Wavesets event interfa
ce,
```

```
// w is the Wavesets2 object which contains all the wavesets data.
// Creation method may change when WavesetsEvent and Wavesets2
// are integrated into a single Wavesets class.
s.waitForBoot {
   ~wsev = WavesetsEvent.read(Platform.resourceDir +/+ "sounds/   a
11wlk01.wav",
        onComplete: {
              w = ~wsev.wavesets;
              b = ~wsev.buffer;
   });
};

w.xings;   // all integer indices of the zero crossings found
w.numXings;     // the total number of zero crossings
w.lengths;      // lengths of all wavesets
w.amps;    // peak amplitude of every waveset
w.maxima;       // index of positive maximum value in every   waves
et
w.minima;            // index of negative minimum value in every
 waveset

w.fracXings;    // fractional zerocrossing points
w.fracLengths;  // and lengths: allows more precise looping   / tun
ing.
w.lengths.plot; // show distribution of lengths
w.amps.plot;

// data for a single waveset: startFrame, length (in frames), durat
ion
w.frameFor(140, 1);

w.maximumAmp(140, 1);    // maximum amplitude of that waveset   or
 group
```

Figure 16.24
A Wavesets object.

Figure 16.26 shows accessing the buffer that a waveset automatically creates, as well as a `SynthDef` to play wavesets with: `buf` is the buffer that corresponds to the waveset, `start` and `length` are the start frame and length (in frames) of the waveset to be played, and `sustain` is the duration for which to loop over the buffer segment. Since the envelope cuts off instantly, one should calculate the precise sustain time and pass it in, as shown in the last section, by using the `frameFor` or `eventFor` method, or

`WavesetsEvent:asEvent`, which provides many conversion methods. (see the `WavesetsEvent`.tutorial for details)

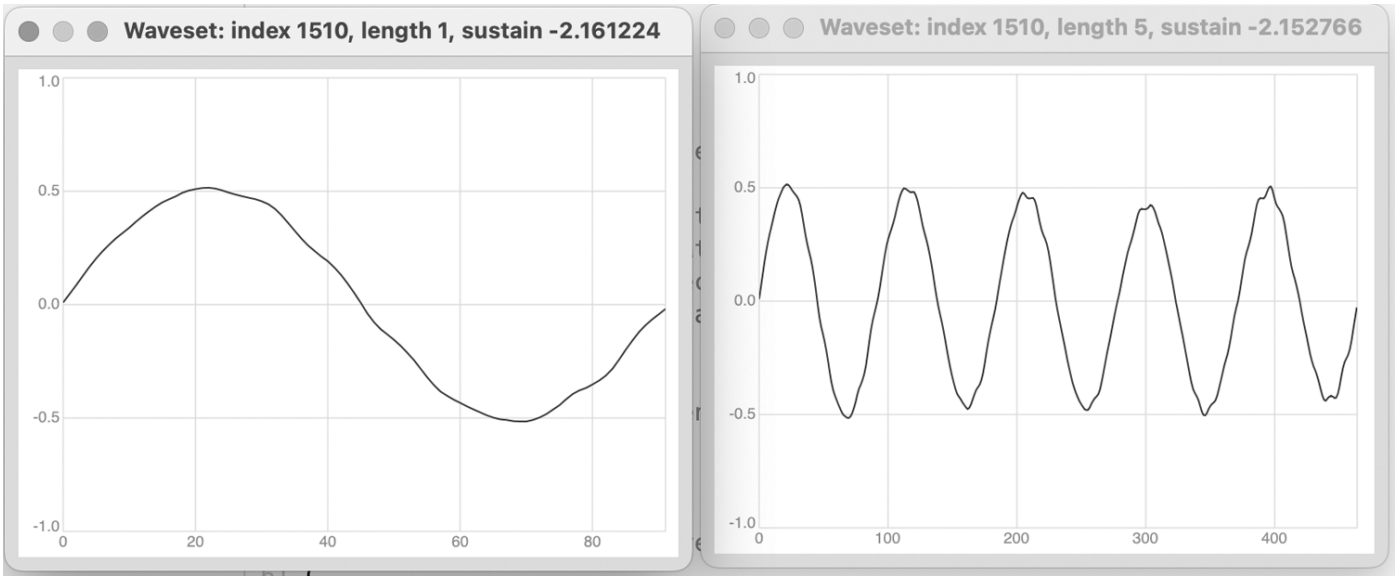


Figure 16.25
A single waveset and a waveset group.

```
// A Synthdef to play a waveset (or group) n times.
(
// Wavesets.prepareSynthDefs loads this SynthDef:
SynthDef(\wvst0, {| out = 0, buf = 0, startFrame = 0,   numFrames =
441, rate = 1, sustain = 1, amp = 0.1, pan,   interpolation = 2 |
   var phasor = Phasor.ar(0, BufRateScale.ir(buf) * rate *   sign(n
umFrames), 0, abs(numFrames)) + startFrame;
   var env = EnvGen.ar(Env([amp, amp, 0], [sustain, 0]),   doneActi
on: 2);
   var snd = BufRd.ar(1, buf, phasor, 1, interpolation) * env;

   OffsetOut.ar(out, Pan2.ar(snd, pan));
}, \ir.dup(9)).add;
)

// make a wavesets play-event with hand-set parameters:
// play from frame 0 to 440, looped for 0.1 secs, so ca 10 repeats.
(instrument: \wvst0, buf: b.bufnum, startFrame: 0, numFrames:   44
0, amp: 1, sustain: 0.1).play;

// get frame data from wavesets with frameFor:
(
```

```
var startFrame, numFrames, sustain, repeats = 20, rate = 1;
#startFrame, numFrames, sustain = w.frameFor(2300, 5);
(
instrument: \wvst0, buf: b.bufnum, amp: 1,
startFrame: startFrame, numFrames: numFrames,
rate: rate,
sustain: sustain * repeats
).postln.play;
)
// or use eventFor:
~wsev.eventFor(startWs: 2300, numWs: 5, repeats: 20, rate: 1).   pu
t(\amp, 0.5).play;

// WavesetsEvent:asEvent prepares a playable event from an inevent,
which supports specifying many waveset parameters. The basic ones a
re:
// start: (waveset index),
// num: number of wavesets to use as group
// repeats: number of repeats of that waveset group
~playme = ~wsev.asEvent((start: 2300, num: 5, repeats: 20,   amp:
 1));
~playme.play;
```

Figure 16.26
Wavesets, buffers, and events.

In order to show Wishart's transforms, we have put a `Pbindef` in figure 16.27, so we can replace patterns or fixed values in it while it is running. It reconstructs the sound file waveset by waveset.

```
// by default, this pattern reconstructs a soundfile segment.
(
Pbindef(\ws1).clear;
Pbindef(\ws1,
   \instrument, \wvst0,
   \startWs, Pn(Pseries(0, 1, w.size-1), 1),
   \numWs, 1,
   \rate, 1,
   \buf, b.bufnum,
   \repeats, 1,
   \amp, 1,
   [\startFrame, \numFrames, \sustain], Pfunc({|ev|
        var start, length, wsDur;
        #start, length, wsDur = w.frameFor(ev[\startWs],   ev[\numW
```

```
s]);
        if (length > 0) {
              [start, length, wsDur * ev[\repeats] / ev[\rate].abs]
      }
   }),
   \dur, Pkey(\sustain)
).play;
)
```

Figure 16.27
A pattern to play wavesets.

Now we can introduce waveset transposition, reversal, time stretching, omission, and shuffling in one-line examples (see figure 16.28).

```
// waveset transposition: every second waveset, half speed
Pbindef(\ws1, \rate, 0.5, \startWs, Pn(Pseries(0, 2, w.size / 2),
1)).play;

// reverse every single waveset
Pbindef(\ws1, \rate, -1, \startWs, Pn(Pseries(0, 1, w.size),   1)).
play;
// reverse every 2 wavesets
Pbindef(\ws1, \numWs, 3, \rate, -1, \startWs, Pn(Pseries(0, 3,   w.
size / 3), 1)).play;
// reverse every 20 wavesets
Pbindef(\ws1, \numWs, 10, \rate, -1, \startWs, Pn(Pseries(0, 10,
 w.size / 10), 1)).play;
// reverse every 200 wavesets
Pbindef(\ws1, \numWs, 30, \rate, -1, \startWs, Pn(Pseries(0, 30,
 w.size / 30), 1)).play;
// reverse every 200 wavesets
Pbindef(\ws1, \numWs, 100, \rate, -1, \startWs, Pn(Pseries(0, 100,
w.size / 100), 1)).play;
// restore
Pbindef(\ws1, \numWs, 1, \rate, 1, \startWs, Pn(Pseries(0, 1,   w.s
ize), 1)).play;

// time stretching
Pbindef(\ws1, \rate, 1, \repeats, 2).play;
Pbindef(\ws1, \rate, 1, \repeats, 4).play;
Pbindef(\ws1, \rate, 1, \repeats, 6).play;
Pbindef(\ws1, \repeats, 1).play;      // restore
```

```
// waveset omission: drop every second waveset
Pbindef(\ws1, \numWs, 1, \freq, Pseq([1, \], inf)) .play;
Pbindef(\ws1, \numWs, 1, \freq, Pseq([1,1, \, \], inf)) .play;
Pbindef(\ws1, \numWs, 1, \freq, Pfunc({if (0.25.coin, 1, \)}))   .p
lay; // drop randomly
Pbindef(\ws1, \numWs, 1, \freq, 1) .play; // restore

// waveset shuffling (randomize waveset order +- 5, 25, 125)
Pbindef(\ws1, \startWs, (Pn(Pseries(0, 1, w.size), 1) + Pfunc(  {5.
rand2})).max(0)).play;
Pbindef(\ws1, \startWs, (Pn(Pseries(0, 1, w.size), 1) + Pfunc(  {2
5.rand2})).max(0)).play;
Pbindef(\ws1, \startWs, (Pn(Pseries(0, 1, w.size), 1) + Pfunc(  {12
5.rand2})).max(0)).play;
```

Figure 16.28
Some of Wishart's transforms.

Waveset harmonic distortion can be realized by playing a chord for each waveset at integer multiples of the rate and appropriate amplitudes. *Waveset interleaving* would also be an easy exercise, as it only requires alternating between waveset objects as sources.

Waveset substitution requires more effort, as one needs to scale the substitute waveform into the time of the original waveform. Substituting a sine wave lets the time structure of the wavesets emerge, especially when each waveset is repeated. Figure 16.29 also considers the amplitudes for each waveset: since the substitute signal is at full volume, scaling it to the original waveset's volume keeps the dynamic contour intact. Finally, different substitute waveforms can have quite different effects.

```
// the waveform to substitute
c = Buffer.alloc(s, 512); c.loadCollection(Signal.sineFill(512,
 [1]));
(
Pbindef(\ws1).clear;
Pbindef(\ws1,
   \instrument, \wvst0,
   \startWs, Pn(Pseries(0, 1, w.size), 5),
   \numWs, 1, \rate, 1,
   \buf, c.bufnum, // sine wave
   \repeats, 1,
   \amp, 1,
   [\startFrame, \numFrames, \sustain], Pfunc({|ev|
        var start, length, wsDur, origRate;
```

```
        origRate = ev[\rate];

        // get orig waveset specs
        #start, length, wsDur = w.frameFor(ev[\startWs],   ev[\numW
s]);

        // adjust rate for different length of substituted wave
        ev[\rate] = origRate * (512 / length);

        // get amplitude from waveset, to scale full volume sine wa
ve
        ev[\amp] = ev[\amp] * w.ampFor(ev[\startWs], ev[\numWs]);

        [0, 512, wsDur * ev[\repeats] / origRate.abs]
    }),
    \dur, Pkey(\sustain)
).play;
)

// clearer sinewave-ish segments
Pbindef(\ws1, \rate, 1, \repeats, 2).play;
Pbindef(\ws1, \rate, 1, \repeats, 6).play;
Pbindef(\ws1).stop;

// different substitution waveforms to try:
c.loadCollection(Signal.sineFill(512, 1/(1..4))); c.plot;   // sawi
sh
// fibonacci overtones
c.loadCollection(Signal.sineFill(512, [1, 1, 1, 0, 1, 0, 0, 1]));
c.plot;
c.loadCollection(Signal.rand(512, -1.0, 1.0)); c.plot; // white   n
oise
c.loadCollection(Signal.sineFill(512, [1])); c.plot;// sine
```

Figure 16.29
Waveset substitution.

Waveset averaging can be realized by adapting this example to play *n* wavesets simultaneously, all scaled to the same average waveset duration, looped *n* times, and divided by *n* for average amplitude.

Waveset inversion, distortion, enveloping, and *time stretching with interpolation* all require writing special `Synthdefs`. Of these, interpolation is sonically most interesting. Although one can imagine multiple ways of interpolating between wavesets,

in the example in the book code repository, the two wavesets are synchronized: the sound begins with *waveset1* at original speed and *waveset2* scaled to the same loop duration, but silent. As the amplitude cross-fades from *waveset1* to *waveset 2*, so does the loop speed, so the interpolation ends with only *waveset 2* at its original speed. The example can be extended to plays a seamless stream of parameterizable interpolations.

Why stick to using wavesets for recognizable sound file transformations? One can also start over with a task that plays a single waveset as a granular stream and extend that gradually. Figure 16.30 begins with a fixed grain repeat rate and starting waveset, later replaced with different values and streams for generating values.

```
// begin very simple with fixed repeat time
(
Tdef(\ws1).set(\startWs, 400);
Tdef(\ws1).set(\numWs, 5);
Tdef(\ws1).set(\repeats, 5);

Tdef(\ws1, {|ev|
   var startFrame, numFrames, wsSustain;
   loop {
   #startFrame, numFrames, wsSustain = w.frameFor(ev.startWs.   nex
t, ev.numWs);

   (instrument: \wvst0, buf: b.bufnum, amp: 1,
        start: startFrame, numFrames: numFrames,
        sustain: wsSustain * ev.repeats;
   ).play;

   0.1.wait;
   }
}).play;
)

Tdef(\ws1).set(\startWs, 420);
Tdef(\ws1).set(\repeats, 3);
Tdef(\ws1).set(\numWs, 2);

// drop in a pattern for starting waveset
Tdef(\ws1).set(\startWs, Pn(Pseries(0, 5, 400) + 500, inf).   asStr
eam);
```

Figure 16.30
Wavesets played with `Tdef`.

In figure 16.31, the wait time is derived from the waveset's duration, and a time gap between wavesets is added. All parameters are called with `.next`, so they can be directly replaced with infinite streams.

```
(
Tdef(\ws1).set(\gap, 3);
Tdef(\ws1, {|ev|
   var startFrame, numFrames, wsSustain, reps;

   loop {
        reps = ev.repeats.next;

        #startFrame, numFrames, wsSustain =
        w.frameFor(ev.startWs.next, ev.numWs.next);

        (instrument: \wvst0, buf: b.bufnum, amp: 1,
              start: startFrame, numFrames: numFrames,
              sustain: wsSustain * reps,
              pan: 1.0.rand2
              ).play;

              // derive waittime from waveset sustain time
              // add gap based on waveset sustain time
              (wsSustain * (reps + ev.gap.next)).wait;
   }
}).play;
)

// experiment with dropping in patterns:
// very irregular gaps
Tdef(\ws1).set(\gap, {exprand(0.1, 20)});
// sometimes continuous, sometimes gaps
Tdef(\ws1).set(\gap, Pbrown(-10.0, 20, 2.0).max(0).asStream);

// random repeats
Tdef(\ws1).set(\repeats, {exprand(1, 20).round});
// randomize number of wavesets per group
Tdef(\ws1).set(\numWs, {exprand(3, 20).round});
Tdef(\ws1).set(\numWs, 3, \repeats, {rrand(2, 5)});
Tdef(\ws1).stop;
```

Figure 16.31
A wait time derived from waveset duration, and a gap added.

A wide range of possibilities opens up here: we could create special orders of the wavesets based, for example, on their lengths or amplitudes; or we could filter all waveset indices by some criterion (e.g., keeping only very soft ones).

For just one example for modifying parameters based on waveset information, see figure 16.32. When we read the waveset lengths as a pitch contour of the file, we can pull all waveset lengths closer to a pitch center, or even invert their lengths around the center to make long wavesets short and vice versa. The `pitchContour` variable determines how drastically this transformation is applied. Waveset omission is also shown.

```
(
Tdef(\ws1).set(\startWs, Pn(Pseries(0, 5, 400) + 500, inf).   asStr
eam);

Tdef(\ws1).set(\gap, 0);
Tdef(\ws1).set(\pitchContour, 0);
Tdef(\ws1).set(\keepCoin, 1.0);
Tdef ('ws1').set('repeats', 5);
Tdef ('ws1').set('numWs', 3);

Tdef(\ws1, {|ev|
   var startFrame, numFrames, wsSustain, reps, numWs,   numFrames2A
vg;
   var squeezer, rate;
   loop {
        reps = ev.repeats.next;
        numWs = ev.numWs.next;

        #startFrame, numFrames, wsSustain =
        w.frameFor(ev.startWs.next, numWs);

        numFrames2Avg = numFrames / numWs / w.avgLength;
        squeezer = numFrames2Avg ** ev.pitchContour.next;
        wsSustain = wsSustain / squeezer;
        rate = 1 * squeezer;

        if (ev.keepCoin.next.coin) {
              (instrument: \wvst0, buf: b.bufnum, amp: 1,
                  start: startFrame, numFrames: numFrames,
                  sustain: wsSustain * reps,
                  rate: rate,
                  pan: 1.0.rand2
```

```
                ).play;
        };

        (wsSustain * (reps + ev.gap.next)).wait;

    }
}).play;
)

// try different pitch Contours:
Tdef(\ws1).set(\pitchContour, 0);    // original pitch

Tdef(\ws1).set(\pitchContour, 0.5); // flattened contour

// waveset overtone singing—all equal duration
Tdef(\ws1).set(\pitchContour, 1.0);

// inversion of contour
Tdef(\ws1).set(\pitchContour, 1.5);
Tdef(\ws1).set(\pitchContour, 2);
Tdef(\ws1).set(\repeats, 3);

// waveset omission
Tdef(\ws1).set(\keepCoin, 0.75);
Tdef(\ws1).set(\keepCoin, 1);

// fade out by omission over 13 secs, pause 2 secs
Tdef(\ws1).set(\keepCoin, Pn(Penv([1, 0, 0], [13, 2])).   asStrea
m).play;

// add a pitch contour envelope
Tdef(\ws1).set(\pitchContour, Pn(Penv([0, 2, 0], [21, 13])).   asSt
ream);
```

Figure 16.32
Add pitch contour and dropout rate.

16.8 Conclusions

The possibilities of microsound as a resource for both sound material and structural ideas are nowhere near being exhausted. One can easily find personal, idiosyncratic ways to create music by exploring recombinations and juxtapositions of synthesis approaches and methods for structuring larger assemblages of microsound events. Due to its generality, SuperCollider supports many different working methods and allows for

flexibly changing direction midway. The pattern library, for instance, offers many ways to create intricately detailed structures that may lead to fascinating microsound textures. Independent of one's aesthetic preferences and of preferred methods for creating music, there are plenty of possibilities for further exploration.

Notes

1. https://microsound.org/.
2. https://www.flucoma.org/.
3. The raw material of the composition "Half-life" (1998–1999) by Roads, was a 1-minute pulsar synthesis train. Section IV ("Organic") of "Clang-tint" (1991–1994) was composed with pulsar synthesis and convolution with various sound sources (e.g., bird, whale, animal, human, and insect sounds)
4. Hecker employed pulsar synthesis on various sound installation works and releases, most notably a CD called *Recordings for Rephlex* (2006).
5. Pietruszewski composed a series of works with the nuPG program, see https://fancyyyyy.bandcamp.com/album/the-new-pulsar-generator-recordings-volume-1 and https://etat.xyz/release/AuditorySieve.
6. https://github.com/marcinpiet/nuPG_1.0.

References

Ariza, Christopher. 2005. "The Xenakis Sieve as Object: A New Model and a Complete Implementation." *Computer Music Journal*, 29(2): 40–60.

Ariza, Christopher. 2009. "Sonifying Sieves: Synthesis and Signal Processing Applications of the Xenakis Sieve with Python and Csound." In *ICMC*.

Born, G., and C. Haworth. 2017. From Microsound to Vaporwave: Internet-Mediated Musics, Online Methods, and Genre. *Music and Letters*, 98(4): 601–647.

Buser, P., and M. Imbert. 1992. *Audition*. Cambridge, MA: MIT Press.

Döbereiner, L. 2011. "Models of Constructed Sound: Nonstandard Synthesis as an Aesthetic Perspective." *Computer Music Journal*, 35(3): 28–39.

Gabor, D. 1947. "Acoustical Quanta and the Theory of Hearing." *Nature,* 159(4044): 591–594.

Haworth, C. 2015. "Sound Synthesis Procedures as Texts: An Ontological Politics in Electroacoustic and Computer Music." *Computer Music Journal*, 39(1): 41–58. https://doi.org/10.1162/COMJ_a_00284.

Hoffmann, P. 2000. "The New GENDYN Program." *Computer Music Journal,* 24(2): 31–38.

Kronland-Martinet, R. 1988. "The Wavelet Transform for Analysis, Synthesis, and Processing of Speech and Music Sounds." *Computer Music Journal,* 12(4): 11–20.

Luque, S. 2006. "Stochastic Synthesis: Origins and Extensions." Master's thesis, Institute of Sonology, Royal Conservatory, The Hague, Netherlands.

Moles, A. A., 1966. *Information Theory and Esthetic Perception*. Champaign: University of Illinois Press.

Moore, B. C. J. 2004. *An Introduction to the Psychology of Hearing,* 5th ed. London: Academic Press

Pietruszewski, Marcin. 2020. "The Digital Instrument as an Artefact." In *From Xenakis's UPIC to Graphic Notation Today,* ed. L. Brümmer, P. Weibel, and S. Kanach (pp. 105–107). Cambridge: Cambridge University Press.

Roads, C. 2001a. *Microsound.* Cambridge, MA: MIT Press.

Roads, C. 2001b. "Sound Composition with Pulsars." *Journal of the Audio Engineering Society,* 49(3): 134–147.

Rocha Iturbide, M. "Les techniques granulaires dans la synthèse sonore." Doctoral thesis, Université de Paris-VIII-Saint-Denis.

Snyder, B., and R. Snyder. 2000. *Music and Memory: An Introduction*. Cambridge, MA: MIT Press.

Sturm, B. L., and J. D. Gibson. 2006. “Matching Pursuit Decompositions of Non-noisy Speech Signals Using Several Dictionaries.” In *Proceedings of ICASSP 2006*, Toulouse, vol. 3, pp. 456–459.

Sturm, B. L., C. Roads, A. McLeran, and J. J. Shynk. 2009. “Analysis, Visualization, and Transformation of Audio Signals Using Dictionary-Based Methods.” *Journal of New Music Research*, 38(4): 325–341.

Sturm, B. L., J. J. Shynk, L. Daudet, and C. Roads. 2008. “Dark Energy in Sparse Atomic Estimations.” *IEEE Transactions on Audio, Speech, and Language Processing,* 16(3): 671–676.

Tremblay, P. A., G. Roma, and O. Green. 2021. “Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit.” *Computer Music Journal*, 45(2): 9–23.

Truax, B. 1988. “Real-Time Granular Synthesis with a Digital Signal Processor.” *Computer Music Journal*, 12(2): 14–26.

Vaggione, H. 2001. “Some Ontological Remarks About Musical Composition Processes.” *Computer Music Journal,* 25(1): 54–61.

Vaggione, H. 1996. “Articulating Micro-Time.” *Computer Music Journal,* 20(2): 33–38.

Wishart, T. 1994. *Audible Design*. London: Orpheus the Pantomime.

Xenakis, I. [orig. 1971] 1992. *Formalized Music: Thought and Mathematics in Music,* rev. and enl. ed. Hillsdale, NY: Pendragon Press.

Xenakis, I., and J. Rahn . 1990. “Sieves.” In *Perspectives of New Music*, pp. 58–67.

17 Alternative Tunings with SuperCollider

Fabrice Mogini

The notions of pitch and tuning are essential to the organization of sound. Our understanding and perception of tuning systems are closely related to the musical instruments being used. An audio programming language transforms the composition, performance, and appreciation of alternative systems of pitch because it can transcend the limitations set by acoustic instruments and human players. SuperCollider (SC) is a powerful tool that gives the opportunity to create and play alternative tuning systems easily in any possible way, and therefore extend compositional scope.

This chapter will demonstrate the ease with which a variety of tuning systems can be explored within the SuperCollider environment. Since tuning is an area of complex and varied experimentation, this chapter will describe several approaches to writing code for alternative tunings with SuperCollider rather than cataloging existing systems. The user can then apply these tools to any specific tuning.

17.1 Standard Tuning: 12-Note Equal Temperament

17.1.1 Tuning

A *tuning* is a set of frequency relationships. We can quantify these relationships with intervals. A fixed frame is needed to help discriminate these intervals. The frame commonly used in most music is the *octave*. The octave is fundamental to many cultures, both ancient and modern (Burns, 1999). A common discussion point for musicians, composers, and scientists has been how to divide the octave. A set of intervals spanning the octave leads to a particular choice of tuning system.

17.1.2 Oscillators and Frequency

We will use oscillators to generate sound and listen to the tunings that we create. Most oscillators in SuperColider require a frequency in hertz (cycles per second):

```
{SinOsc.ar(523.2511306012, 0, 0.5)}.play;
```

Although we can specify frequencies directly in hertz, it is often easier to deal with notes that point at a position in the temperament or scale used.

17.1.3 MIDI Notes

MIDI notes are based on the standard 12-note, equal-tempered tuning system, in which each octave is split into 12 equal steps. The message `midicps` in SuperCollider converts MIDI notes to cycles per second. Using the message `midicps` is an easy way to access different degrees from the 12-note equal temperament without having to know what frequency it corresponds to:

```
60.midicps // frequency in hertz for the first note, middle C
61.midicps // C sharp above middle C
72.midicps // C in the next higher octave
```

Rather than having to know the exact frequency, we can just choose the MIDI note number and use `midicps` to return the corresponding frequency in hertz:

```
{SinOsc.ar(72.midicps, 0, 0.5)}.play;
```

17.1.4 Midiratio

Some unit generators in SuperColider require a ratio value; for instance, a rise of an octave is represented by the number 2 (to create a frequency twice as fast). This is the case with sample playback using `PlayBuf`, which has a `rate` argument representing the speed at which a sample is played relative to a baseline of its original pitch. Speed being related to pitch, we can use this rate argument to modify the sample's pitch. (See figure 17.1.) Here we use `midiratio`, which converts an interval in semitones to a ratio:

```
// first note
0.midiratio
// second note
1.midiratio
// first note, one octave up
12.midiratio

// read a whole sound into memory
b = Buffer.read(s, Platform.resourceDir +/+"sounds/a11wlk01   .wa
v"); // remember to free the buffer later.
(
SynthDef("help_PlayBuf", {arg out=0, bufnum=0, rate=1;
```

```
    Out.ar(out,
          Pan2.ar(
                PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum)*rate,
loop: 1),
                0)
    )
}).add;
)
p=Synth(\help_PlayBuf, [\rate, 0.midiratio, \bufnum, b]);   // orig
inal pitch
p.set(\rate, 12.midiratio);  // one octave up
p.set(\rate, 7.midiratio);   // seven semitones up (fifth   interva
l)

(
b.free;
p.free;
)
```

Figure 17.1
Example of `PlayBuf` with `midiratio`.

Multiplying the ratio by a constant root, we obtain the frequencies for each degree:

```
(0.midiratio)*440
(1.midiratio)*440
(12.midiratio)*440
{SinOsc.ar((0.midiratio)*440, 0, 0.5)}.play;
{SinOsc.ar((7.midiratio)*440, 0, 0.5)}.play;
// third note of the chromatic scale plus a quarter tone
{SinOsc.ar((2.5.midiratio)*440, 0, 0.5)}.play;
```

We can also write this in a different way, using a function.

All the following examples run sequentially. Stop the sound before playing the next example:

```
f = {|degree, root = 440| (degree.midiratio)*root};
{SinOsc.ar(f.(0), 0, 0.5)}.play;
{SinOsc.ar(f.(7), 0, 0.5)}.play;
{SinOsc.ar(f.(2.5), 0, 0.5)}.play;
```

We can change the root if necessary:

```
{SinOsc.ar(f.(0, 261.5), 0, 0.5)}.play;
{SinOsc.ar(f.(7, 261.5), 0, 0.5)}.play;
{SinOsc.ar(f.(2.5, 261.5), 0, 0.5)}.play;
```

In this next function, the root can be given as a MIDI note:

```
f = {|degree, root = 60|
(degree.midiratio)*root.midicps;
};
f.(0, 65);
f.(1, 65);
f.(12, 65);
```

17.1.5 Cents

Cents is a measure often used for fine-tuning intervals. Although it might seem that SuperCollider does not support cents directly, MIDI values can describe cents because `midicps` accepts decimals. We can obtain frequencies between two semitones:

```
60.midicps;       // middle C
60.5.midicps;    // middle C and a quarter tone
61.midicps;      // C#
```

One octave is divided into 1,200 cents, and a semitone equals 100 cents. There are 100 cents between `60.midicps` and `61.midicps`:

```
60.01.midicps    // middle C plus 1 cent
60.5.midicps     // middle C plus 50 cents or a quarter tone
60.99.midicps;   // just 1 cent below C#
61.midicps;       //C#
// C on the left and D# plus a quarter tone on the right
{SinOsc.ar([60, 63.5].midicps, 0, 0.5)}.play;
```

17.1.6 Pbind and the Pitch Model

There are several ways to access different notes in the 12-note equal temperament using `Pbind`: modal scale degrees, equal-division note values, MIDI note values, or frequencies in hertz. (See also the Help files StreamsPatternsEvents5, Event and PG_07_Value_Conversions.)

The key `\midinote` is similar to `midicps` but is used exclusively by the `Pitch` model of an `Event`. (See figure 17.2.)

```
(
Pbind(
    \midinote, Pseq([0, 2, 3, 5, 7]+60, inf),
    \dur, 0.3
).play
)
```

Figure 17.2
Example of `Pbind` with `\midinote`.

You can alter the pitch of `\midinote` in cents:
62.5 is equivalent to 62 + 50 cents.
62.25 is equivalent to 62 + 25 cents. (See figure 17.3.)

```
(
Pbind(
    \midinote, Pseq([0, 2, 3, 5.25, 7.5]+60, inf),
    \dur, 0.3
).play
)
```

Figure 17.3
Example of `Pbind` with `\midinote` and cents.

Here, `\note` seems quite similar to `\midinote`, but its octave range is selected by the `\octave` argument. (See figure 17.4.)

```
(
Pbind(
    \note, Pseq([0,2,4,5,7,9,11,12], inf),
    \dur, 0.3,
    \octave, 5
).play
)
```

Figure 17.4
Example of `Pbind` with `\note`.

The argument `\degree` points at a particular position in a scale. In figure 17.5, the scale argument is not specified and, by default, is `[0, 2, 4, 5, 7, 9, 11]`, which corresponds to a major scale in 12-note equal temperament.

```
(
Pbind(
    \degree, Pseq([0,1,2,3,4,5,6,7], inf),
    \dur, 0.3,
    \octave, 5
).play
)
```

Figure 17.5
Example of `Pbind` with `\degree`.

Note that the `\scale` argument can also be changed. If the scale was set as the chromatic scale, we could use `degree` to access all the notes of the standard temperament, although since `\note` already does this, it is much more typical to exploit unequal scales from major to Hungarian minor and beyond. (See figure 17.6.)

```
(
Pbind(
    \degree, Pseq([0,1,2,3,4,5,6,7], inf),
    \dur, 0.3,
    \octave, 5,
    \scale, (0..11)
).play
)
```

Figure 17.6
Example of `Pbind` with `\degree` and `\scale` (chromatic scale).

We can alter the pitch of these degrees in cents:
2.1 is equivalent to 3, the next degree, or 2 + 100 cents.
2.05 is equivalent to 2 + 50 cents.
Note that this is different from the way we altered cents earlier when using `\midinote`. Figure 17.7 gives an example using `degree` and cents.

```
(
Pbind(
    \degree, Pseq([0, 2, 2.1, 2.05], inf),
    \dur, 0.3,
    \scale, (0..11),
    \octave, 5
).play
)
```

Figure 17.7
Example of `Pbind` with `\degree` and `\scale` and cents.

17.2 Other Equal Temperaments

17.2.1 `\stepsPerOctave`

The `Pitch` model uses `\stepsPerOctave`, which is 12 by default, meaning we are in a 12-equal-note system. Just specify a new number of steps per octave and start exploring. Let us start by playing 7 equal notes per octave. We will be using `Pwhite` to randomize the selection of notes from that tuning and `\sustain` to create polyphonic layers. (See figure 17.8.)

```
(
Pbind(
    \note, Pwhite(-6,9),
    \dur, 0.3,
    \sustain, 1.1,
    \stepsPerOctave, 7
).play
)
```

Figure 17.8
Example of `Pbind` with `\stepsPerOctave`.

17.2.2 \scale

When using a large number of notes per octave, it is a good idea to create a mode using \scale. (See figures 17.9 and 17.10.)

```
(
e = Pbind(
    \degree, Pwhite(-3, 7),
    \dur, 0.25,
    \stepsPerOctave, 21,
    \sustain, 1.1,
    \scale, [0, 4, 8, 11, 14, 17]
).play;
)
```

Figure 17.9
Example of `Pbind` with `\stepsPerOctave` and `\scale`.

```
(
// previous example should still be running
e.stream = Pbind(
    \degree, Pwhite(-3, 7),
    \dur, 0.25,
    \stepsPerOctave, 21,
    \sustain, 1.1,
    \scale, [0, 3, 5, 8, 10, 13]
).asStream;
)
```

Figure 17.10
Example of changing mode using `\scale`.

17.2.3 Calculation of Equal Temperaments

When we don't use existing functionality from `midicps` to `Pbind`, it is helpful to know how to calculate equal temperaments for more flexibility. We can start by dividing a value that represents the octave into equal parts. When we divide 1, the space that represents 1 octave, by the number of steps per octave, we obtain the smallest interval in linear terms: 1/12. To get the third note, we multiply this interval by 3: 3 * (1/12). The third note also can be divided by steps per octave: 3/12.

However, because frequencies grow exponentially, the space in hertz between each note of the tuning is in fact getting larger and larger, so we need to use the power of 2.

```
// 2.pow(degree/stepsPerOctave); so the third note of the   12-equa
l-note tuning system
2.pow(3/12);
```

Figure 17.11 gives an example of calculating intermediate step ratios automatically based on the octave ratio 2. Pseudooctaves are derived by substituting values other than 2: `3.pow(degree/13)` would give the Bohlen-Pierce scale based on 13 equally spaced notes over a tritave (octave and a fifth) basic ratio; more examples will follow later in this discussion.

```
(
var stepsperoctave = 3;
Array.fill(stepsperoctave, {arg i; 2.pow(i/stepsperoctave)});
)

// Using a function to calculate the value at a chosen degree
(
```

```
f = {|degree, steps|
   2.pow(degree/steps)
};
)
// degree 0
f.(0, 3);
// degree 1
f.(1, 3);
// degree 2
f.(2, 3);

// The function is modified to multiply the value by a root frequen
cy in Hertz
(
f = {|degree, steps, root=440|
   2.pow(degree/steps)*root
};
)
// 12 notes per octave, degrees 0, 1 and 12
f.(0, 12)
f.(1, 12)
f.(12, 12)

// 14 notes per octave, degrees 0, 1, 12 and 14
f.(0, 14)
f.(1, 14)
f.(12, 14)
f.(14, 14)
```

Figure 17.11
Example of a simple tuning of three equal notes per octave.

17.2.4 Using degreeToKey

Next, consider that `degreeToKey` has a `stepsPerOctave` argument but is designed for modal control. As we saw earlier with `\scale` and `Pbind`, a mode is a set of notes from the current tuning system. All the notes from the tuning can be accessed by specifying a chromatic scale that contains all degrees within the octave:

```
0.degreeToKey((0..13), 14)
1.degreeToKey((0..13), 14)
15.degreeToKey((0..13), 14)
```

We can limit the scale to a mode:

```
0.degreeToKey([0, 3, 5, 9], 14)
1.degreeToKey([0, 3, 5, 9], 14)
15.degreeToKey([0, 3, 5, 9], 14)
```

Note that `degreeToKey` only indicates the degree in the tuning according to the scale, but it doesn't actually calculate the frequency for us:

```
// getting the degree
a = 0.degreeToKey([0, 3, 5, 6], 12)
// calculating the frequency
2.pow(a/12)*440
// another degree in the scale
b = 4.degreeToKey([0, 3, 5, 6], 12)
// calculating the frequency
2.pow(b/12)*440
```

Here, `degreeToKey` is a UGen that converts a signal to modal pitch. It is different from the message `degreeToKey` that was discussed earlier. Even though we won't cover it in this chapter, it is a useful tool which supports any equal temperament, and therefore it is worth mentioning.

17.3 Unequal Divisions of the Octave

We will now work with frequency ratios more directly.

17.3.1 Custom Unequal Division of the Octave

Any array in SuperCollider can be filled with values between 1 and 2; this array represents the space across one octave:

```
a = [1, 1.030303030303, 1.0606060606061, 1.1212121212121,   1.21212
12121212, 1.3636363636364, 1.6060606060606]
```

To obtain frequencies, we can multiply the array by a constant root: a = a*220. With `Pbind`, we need to use `\freq` because we have calculated the frequencies ourselves. (See figure 17.12.)

```
(
SynthDef("tone2", {arg freq = 440, amp=0.5, gate=1, envdur=1.5;
   var sound, env;
   env = EnvGen.kr(Env.perc(0.01, envdur), doneAction:2);
   sound = Pan2.ar(SinOsc.ar(freq, 0, amp)*env, 0);
```

```
    Out.ar(0, sound);
}).add;
a = [1, 1.030303030303, 1.0606060606061, 1.1212121212121,   1.36363
63636364, 1.6060606060606, 2] * 220;
)

(
// Play the all the notes of the tuning
e = Pbind(
    \freq, Pseq (a, inf),
    \dur, 0.2,
    \amp, 0.5,
    \sustain, 0.6,
    \instrument, \tone2
).play
)

// Choose the notes randomly
(
e.stream = Pbind(
    \freq, Pn(Prand (a, 1)),
    \dur, 0.2,
    \amp, 0.5,
    \sustain, 0.6,
    \instrument, \tone2
).asStream
)
```

Figure 17.12
Example of `Pbind` with unequal octave divisions for `\freq`.

The `linlin` message is a useful method for mapping values to a different range and for adapting your own series to the system you are working with: `this.linlin(inMin, inMax, outMin, outMax, clip)`:

```
a = [1, 2, 3, 5, 8, 13, 21]
```

We can map this array to values between 1 and 2 (the octave) with similar proportions:

```
b = a.linlin(1, 34, 1, 2);
```

We obtain the unequal tuning that was calculated in the earlier example with `Pbind`:

```
a = [1, 1.030303030303, 1.0606060606061, 1.1212121212121,   1.21212
12121212, 1.3636363636364, 1.6060606060606, 2]
```

17.3.2 Ratios

The octave (2/1) is the standard space that can be deconstructed into different proportions. This is different from the equal temperament system, where we had steps of the same size.

SuperCollider will calculate any fraction directly: evaluating `5/4` returns 1.25. We can also simplify the terms of a fraction with `asFraction`. Here, `(10/8).asFraction` returns `[5, 4]`.

17.3.3 Just Tuning

This ratio-based system is founded on the premise that dissonance is associated with more complex ratios (such ratios tend to beat more). The simpler the ratios, the more consonant the intervals in terms of purity of sound. Consonance is a priority in just intonation, even though this can limit modulatory freedom in changing key. (Different keys have very different characters, whereas in 12-note equal temperament, all keys have the same character.)

A collection of low whole-number ratio intervals forming a diatonic just scale is `a = [1/1, 9/8, 5/4, 4/3, 3/2, 5/3, 15/8, 2/1]`. We can use this array by multiplying it by a constant root frequency: `a = [1/1, 9/8, 5/4, 4/3, 3/2, 5/3, 15/8, 2/1] * 440`.

17.3.4 Odd Limit

The *odd limit* is the set of ratios in which odd number factors are inferior or equal to the chosen number (the limit). The *three-limit tonality* is a set of ratios in which the odd numbers are inferior or equal to 3: 1/1, 4/3, 3/2. The *five-limit tonality* is similar to just tuning: 1/1, 6/5, 5/4, 4/3, 3/2, 8/5, 5/3. The *seven-limit tonality* is 1/1, 8/7, 7/6, 6/5, 5/4, 4/3, 7/5, 10/7, 3/2, 8/5, 5/3, 12/7, 7/4. (A musical example appears in figure 17.13.)

```
(
~rationames = [1/1, 8/7, 7/6, 6/5, 5/4, 4/3, 7/5, 10/7, 3/2,   8/5,
5/3, 12/7, 7/4];
~scale = [0, 3, 5, 8, 10, 12];
e = Pbind(
   \freq, Pseq([
        Pfunc({
              (~rationames.wrapAt(~scale).[~scale.size.rand])*440
        })
```

```
    ],inf),
    \dur, 0.25,
    \amp, 0.5,
    \instrument, \tone2
).play; // returns an EventStream
)
// set a new scale
~scale = [0, 2, 5, 7, 9, 11];
~scale = [0, 1, 3, 5, 6, 8, 9];
~scale = [0, 3, 5, 8, 10, 12];
```

Figure 17.13
Example of using odd-limit ratios with sound.

17.3.5 Tonality Diamond

This is a way of ordering ratios belonging to an *n*-limit tonality set. Harry Partch (1974) experimented with these low-integer rational ratios and built a system in which both simple and complex ratios coexist. One of them is his famous 43-note ratio tuning. (See figure 17.14.)

```
(
var n, buts, synths, ratios, rationames;
w = Window("tonality diamond", Rect(200,500,420,150));
w.view.decorator = FlowLayout(w.view.bounds);
rationames=[
    "7/4", "3/2","5/4","1/1",
    "7/5","6/5","1/1","8/5",
    "7/6","1/1","5/3","4/3",
    "1/1","12/7","10/7","8/7"
];

n=rationames.size;

n.do({|i|
    Button(w, Rect(20,20+(i*30),100,30))
    .states_([[rationames[i], Color.black,
         if((rationames[i])=="1/1", {Color.red},{Color.yellow})
    ]

    ])
    .action_({arg butt;
         Synth(\tone2, [\freq, ((rationames[i]).interpret)*440]);
```

```
    })
});
w.front;
)
```

Figure 17.14
Code for a tonality diamond with a SuperCollider GUI.

Note that this arrangement helps in understanding how neighboring ratios are related. For instance, 7/4 and 12/7 are close, but they are dissonant when played together. However, 7/4 is harmonious with its adjacent ratio 3/2, and 12/7 is harmonious with 10/7. (See figure 17.15.)

Figure 17.15
Picture of a tonality diamond with the SuperCollider GUI.

17.4 Polytunings (Mixed Tunings)

17.4.1 Simple Mixing

It is possible to mix different tunings in SuperCollider. The process is straightforward. (See figure 17.16.)

```
(
a = Pbind(
    \degree, Pwhite(0, 12),
    \dur, 0.5,
    \octave, 5,
    \amp, 0.4,
    \stepsPerOctave, 12,
    \instrument, \tone2
```

```
);
b = Pbind(
    \degree, Pwhite(0, 14),
    \dur, 0.25,
    \octave, 4,
    \amp, 0.4,
    \stepsPerOctave, 14,
    \instrument, \tone2
);
Ppar([a, b]).play;
)
```

Figure 17.16
Example of 12 and 14 equal notes per octave mixed together.

Polytunings (Mogini, 2000) are a system that uses different tunings simultaneously. Common frequencies are used as pivots to connect these tunings. In order to choose the best pivot frequencies, we select pairs of frequencies that are close enough to one another. This system can connect 2 or more tunings. Because so many sounds are now available within an octave, we have found a way of accessing and ordering what I call the "total pitch field."

17.4.2 Organizing Pitch

It is important at this stage to create ways of ordering the choice of notes so as to control the level of consonance/dissonance. Finding nodes common to both tunings can be done by comparing the 2 scales linearly:

```
// 2 different equal tunings expressed linearly
a = Array.fill(12, {|i| (1/12)*(i)});
b = Array.fill(14, {|i| (1/14)*(i)});
a.sect(b);
```

The roots, 0 and 0.5, are common to both tunings. We can give priority to these common nodes.

Find in which position the value 0.5 is in each array as follows:

```
a.do({|item, index| if(item == 0.5, {index.postln})});
// returns 6
b.do({|item, index| if(item == 0.5, {index.postln})});
// returns 7
```

Figure 17.17 presents a musical example of this.

```
(
a = Pbind(
    \degree, Pfunc({
         [
               [0, 6, 12].choose, 12.rand
         ].choose;
    }),
    \dur, 0.5,
    \octave, 4,
    \amp, 0.4,
    \stepsPerOctave, 12,
    \instrument, \tone2
);
b = Pbind(
    \degree, Pfunc({
         [
               [0, 7, 14].choose, 14.rand
         ].choose;
    }),
    \dur, 0.25,
    \octave, 5,
    \amp, 0.3,
    \stepsPerOctave, 14,
    \instrument, \tone2
);
Ppar([a, b]).play;
)
```

Figure 17.17
Example of two tunings organized by their most common notes.

17.4.3 Tolerance Threshold

In a musical context, it is possible to imagine fast passing notes that are dissonant. However, in the slow and regular parts of the music, the sense of dissonance would be increased. The notion of "in tune" can be alternated with "out of tune" if done at the right time and place. One could change the tolerance threshold (limits of the accepted dissonance) according to the compositional context (Mogini, 2000). This system, originally developed by the author, was ported to a GUI by Nick Collins in 2000 in order to have fast control over the tolerance threshold in real time. Below is the author's way of calculating near-tones and the tolerance threshold in order to define which notes are considered common to both tunings. (See figure 17.18.)

```
(
~tolerance={|a, b, t, max|
   var c, d;
   c=[];
   d=[];
   a.do({|aitem, aindex|
        b.do({|bitem, bindex|
              var x;
              x = (aitem-bitem).abs;
              if ((x > t) && (x < max),
                  {
                  c=c.add(aindex);
                  d=d.add(bindex);
                  //[aitem, bitem].post;" out of tune ".post; [aind
ex, bindex].postln;
                  //" ".postln;
              })
        })
   });
   [(0..a.size).difference(c), (0..b.size).difference(d)];
};
)

(
// use the function function with two tunings
var mintreshold, maxtreshold, int;

// two different equal tunings expressed linearly
a=Array.fill(12, {|i| (1/12)*(i)});
b=Array.fill(21, {|i| (1/21)*(i)});

int=1/21; // smallest interval
mintreshold=int*0.15;
maxtreshold=int*0.85;
/*
intervals inferior to mintreshold are in tune
intervals between mintreshold and maxtreshold are out of tune
intervals superior to maxtreshold are in tune
*/

// print a list of notes from the two tunings which form a dissonan
t interval
```

```
~tolerance.value(a, b, mintreshold, maxtreshold);
)
```

Figure 17.18
Example of calculating near-tones, setting a tolerance threshold.

This simple function has enabled us to define an area in which 2 notes from different tunings are *in tune*. This notion of what is *in tune* is subjective and can be changed by the user. This example has been simplified for a greater understanding. It would be easy to rewrite the function so that we can change the `\degree` argument directly or even control the `\scale` argument in real time. (See figure 17.19.)

```
(
a=Pbind(
   \degree, Pfunc({
        // notes which clash with the other tuning have been remove
d
        [0,4,8,12].choose
   }),
   \dur, 0.5,
   \octave, 5,
   \amp, 0.4,
   \sustain, 0.85,
   \stepsPerOctave, 12,
   \instrument, \tone2
).play;
b=Pbind(
   // notes which clash with the other tuning have been removed
   \degree, Pfunc({
        [0,7,14,21].choose
   }),
   \dur, 0.25,
   \octave, 4,
   \amp, 0.35,
   \sustain, 0.85,
   \stepsPerOctave, 21,
   \instrument, \tone2
).play;
)

(
a.stream=Pbind(
   // introducing more notes from that tuning after having changed
 the threshold
```

```
    \degree, Pfunc({
          [0, 1, 4, 7, 8, 9, 9, 12] .choose
    }),
    \dur, 0.75,
    \octave, 5,
    \amp, 0.4,
    \sustain, 0.85,
    \stepsPerOctave, 12,
    \instrument, \tone2
).asStream;
)
```

Figure 17.19
Example changing the number of common notes in real-time.

Note that we are using results from figure 17.18 to choose the degrees.

17.5 Beyond the Octave Division

Intervals other than the octave can be equally divided and then represent a tuning system.

17.5.1 Dividing Any Interval into Equal Notes

We are now going to construct an array of equal values within a range that stretches from 1 to a value greater than 2. We could of course also design an array that has a different range. In fact, any range and number of steps can be used. We already know how to divide an interval of 1 (from 1 to 2) into 12 equal parts mapped to represent an exponential progression.

```
//   Function to generate equal-note temperaments
f = {|steps| Array.fill(steps, {|i| 2.pow(i/steps)})};
//   Calculation of the 12-equal-note temperament
x = f.(12);
```

As we have seen in section 17.3, we can use `linlin` to map an array to a different range while keeping the same proportions between each item. For instance, we can map the tuning to a new range beyond an octave:

```
y = x.linlin(1, 2, 1, 2.25);
```

We have a new array of 12 equal steps and similar proportions with a new range from 1 to 2.25. We can now multiply our array by a constant root frequency: `a =`

`y*220;`. (See figure 17.20.)

```
(
f = {|steps| Array.fill(steps, {|i| 2.pow(i/steps)})};
// Calculation of the twelve equal-note temperament
x = f.(12);
// mapping the tuning to a new range beyond an octave
y = x.linlin(1, 2, 1, 2.25);
// multiplying by a root frequency
a=y*440;
Pbind(
   \freq, Pfunc({a.choose}),
   \dur, 0.25,
   \octave, 5,
   \amp, 0.5,
   \sustain, 1.1,
   \instrument, \tone2
).play
)
```

Figure 17.20
Example of 12 equal-note division beyond the octave.

We can also modify the number of steps per octave to obtain similar results. (See table 17.1 and figure 17.21.)

Table 17.1
Example of Tunings by W. Carlos.

Tuning name	Steps per octave	Cents per interval
Alpha	15.385	78
Beta	18.809	63.8
Gamma	34.188	35.1

```
(
Pbind(
   \degree, Pwhite(0, 18),
   \dur, 0.3,
   \sustain, 1.0,
   \amp, 0.5,
   \sustain, 1.1,
   \instrument, \tone2,
   \stepsPerOctave, 18.809
```

```
).play;
)
```

Figure 17.21
Example of Tuning by W. Carlos with SuperCollider.

17.5.2 Dividing Any Interval into Unequal Notes

As we have done earlier with unequal division of one octave, we will assume that the space between each item in the arrays presented in the following examples is already calibrated to an exponential progression to work when directly multiplied by a constant frequency.

Any array in SuperCollider can be filled with values between 1 and a value other than 2 to define the range of our tuning.

```
a = [1, 1.09375, 1.1875, 1.28125, 1.375, 1.46875, 1.5625,   1.6562
5]
```

To obtain frequencies, we multiply the array or each item by a constant root: `b = a*220`. (See figure 17.22.)

```
(
a=[1, 1.09375, 1.1875, 1.28125, 1.375, 1.46875, 1.5625, 1.65625];
b=a*440;
e=Pbind(
   \freq, Pseq(b, inf),
   \dur, 0.2,
   \amp, 0.5,
   \instrument, \tone2,
   \sustain, 0.6
).play
)

// play in a different order
(
e.stream=Pbind(
   \freq, Pn(Pshuf(b, 1)),
   \dur, 0.2,
   \amp, 0.5,
   \instrument, \tone2,
   \sustain, 0.6
).asStream
)
```

Figure 17.22
Example of `Pbind` with unequal divisions below an octave.

Further accessing the degrees, we can also extend the range. Simply multiply our array by the maximum value of its range to obtain the next series. Again, we could have used `linlin` to calculate the next array.

```
a = [1, 1.25, 1.5, 1.75, 2, 2.25];
// 2.5 is the next value, the range is really [1, 2.5] rather than
 [1, 2]
// next series starts from 2.5
b = a*2.5; // returns:
[2.5, 3.125, 3.75, 4.375, 5, 5.625]
```

17.5.3 Pattern-Based Tunings

These tunings consist of unequal notes and no octave. However, because the proportions between these notes are repeated, a sense of tuning is created. For Burns (1999) a tuning system does not operate independent from other compositional factors. Melodies are perceived in gestalts or patterns rather than as a succession of intervals. To incorporate this idea, I worked on a tuning system based on melodic permutations. The intervals do not fit in the frame of an octave, but the way they go beyond the octave limit is dictated by melodic permutations.

For the next example, I have used a limited set of intervals of different sizes that are not multiples of each other. I would not classify these intervals among ratios, since ratios are still deduced from the octave.

These intervals have different sizes, but each frequency available in the tuning is not fixed because it is not based on a frame, instead being calculated from the last note. To decide on these intervals I have used unequal parts: `[1.1428, 1.36, 1.26]`. Each new frequency in the pattern yields a node from which the melodic development takes place. (See figure 17.23.)

```
(
// F. Mogini pattern-based Tuning—2000.
x = 880;

Pbind(
   \freq, Pn(
        Plazy({
              if(x<=150, {x=x*2});
              if(x>=2000, {x=x/2});
                    x=[
```

```
                    x*[1.1428, 1.36, 1.26].choose,
                        x/[1.1428, 1.36, 1.26].choose

                ].choose
        })

    ),
    \dur, 0.14,
    \sustain, 0.8,
    \cutoff, Pfunc({1.0.rand})
).play;
)
```

Figure 17.23
Example of a pattern-based tuning.

17.6 Tuning Systems and Harmonics

17.6.1 Harmonic Series

Pure sounds with a simple, singular sound wave are rare in nature. Most musical sounds contain partials that are multiples of the fundamental frequency. As the partials fuse for pitch perception, we tend to hear only this single fundamental frequency, but with a particular timbre which is the result of the harmonic complex. (See figure 17.24.)

```
a = (1..16) * 100;
(
e = Pbind(
    \freq, Pseq (a, inf),
    \dur, 0.2
).play
)

// a beautiful tuning system can be created from the harmonic serie
s.
(
e.stream = Pbind(
    \freq, Pn(Pshuf (a, 1)),
    \dur, 0.2,
    \sustain, 0.8
).asStream
)
```

Figure 17.24
Calculation of the first 16 harmonics for a root note of 440 Hertz.

It is possible to keep dividing some of the notes by 2 so as to rearrange all the notes inside one octave. (See figure 17.25.)

```
a = (1..11);
(
a.size.do({|i|
   var x = a[i];
   while({x > 2},{x = x/2});
   a.put(i, x)
});
)

b=a.asSet.asArray.sort;
(
e = Pbind(
   \freq, Pn(Pshuf (b * 440, 1)),
   \dur, 0.2,
   \sustain, 0.8
).play
)
```

Figure 17.25
Rearranging the first 16 harmonics within one octave.

Different arrangements are possible as well. For instance, you can keep certain notes in the first octave and others, the more dissonant ones, in the second octave. (See figure 17.26.)

```
a = (1..8);
b = (9..16);
(
a.size.do({|i|
   var x = a[i];
   var y = b[i];
   // harmonics below 8 remain in the first octave
   while({x > 2},{x = x/2});
   // harmonics above 9 remain in the second octave
   while({y > 4},{y = y/2});
   a.put(i, x);
   b.put(i, y);
```

```
});
[a, b] // see the two arrays in the post window
)
c = (a ++ b).asSet.asArray.sort;
(
e = Pbind(
   \freq, Pn(Pshuf (c*200, 1)),
   \dur, 0.2,
   \sustain, 1.1
).play
)
```

Figure 17.26
Different arrangements for the first 16 harmonics.

We could invert the process and get a slightly different tuning, in which the most dissonant harmonics are in the bass and the most consonant ones are in the highest range.

17.6.2 Creating Artificial Partials for Alternative Equal Temperaments

Inspired by natural harmonics, we can create harmonics specially designed for the tuning that we want to use. Of course, these aren't real harmonics, but this technique can bring a new dimension, timbre, to our exploration of alternative tunings (see also Sethares 2005). Each complex tone will be the result of all these frequencies played together with different amplitudes, a technique similar to wave table synthesis. This simple example is just a starting point. More interesting sounds could result, for instance, from using louder amplitudes for even multiples of the root frequency.

The function can further be rewritten to place the most dissonant notes in the highest range. (See figure 17.27.)

```
(
// a function to expand the tuning from one octave to four octaves
~harmsfunc = {arg stepsperoctave=7;
   var harms;
   // calculate each note from the tuning
   harms = Array.fill(stepsperoctave, {arg i; 2.pow(i/   stepsperoc
tave)});
   harms.size.do({|i|
        if (0.6.coin, {
             // multiply some of the notes to create higher harmon
ics
             harms.put(i, (harms[i])*[1,2,4,8].choose)
```

```
        })
    });
    harms.sort;
};
)

// create an array of virtual harmonics, seven equal-note temperament
~harms = ~harmsfunc.value(7);

(
// send a synth definition with some partials and the current value of ~harms

SynthDef(\partials, {arg out=0, freq = 360, gate = 1, pan,   amp=0.8;
    var sound, eg, fc, osc, a, b, w;
    var harms, amps;

    // use the harmonics previously calculated
    harms = ~harms;
    //   create new amplitudes for each harmonic
    amps = Array.fill(harms.size,{1.0.rand}).normalizeSum*0.1;

    osc = Array.fill(harms.size, {|i|
        SinOsc.ar(freq * harms[i], 0, amps[i]);
    })++[SinOsc.ar(freq, 0, amp*(0.5.rand+0.2)), SinOsc.   ar(freq*2, 0, amp*(0.5.rand+0.15))];

    eg = EnvGen.kr(Env.asr(0.02,1,1), gate, doneAction:2);

    sound = Pan2.ar(eg * Mix.ar(osc), pan);
    Out.ar(0, sound);
}).add;
)

(
e = Pbind(
    \instrument, \partials,
    // frequencies are rpeated so we can notice the effect of harmonics
    \degree, Pseq([0,1,2,3,4,5,6,7],inf),
    \dur, 0.25,
```

```
    \stepsPerOctave, 7,
    \octave, 4,
    \pan, Pfunc({0.5.rand2})
).play;
)

// Send the SynthDef function again to obtain new amplitudes for ea
ch harmonic
(
// send a synth definition with some partials and the current value
of ~harms

SynthDef(\partials, {arg out=0, freq = 360, gate = 1, pan,   amp=0.
8;
    var sound, eg, fc, osc, a, b, w;
    var harms, amps;

    // use the harmonics previously calculated
    harms = ~harms;
    //  create new amplitudes for each harmonic
    amps = Array.fill(harms.size,{1.0.rand}).normalizeSum*0.1;

    osc = Array.fill(harms.size, {|i|
         SinOsc.ar(freq * harms[i], 0, amps[i]);
    })++[SinOsc.ar(freq, 0, amp*(0.5.rand+0.2)), SinOsc.   ar(freq*
2, 0, amp*(0.5.rand+0.15))];

    eg = EnvGen.kr(Env.asr(0.02,1,1), gate, doneAction:2);

    sound = Pan2.ar(eg * Mix.ar(osc), pan);
    Out.ar(0, sound);
}).add;
)

// re-evalute the function to create new harmonics (update the Synt
hDef afterwards)
~harms = ~harmsfunc.value(7);
// Send the SynthDef above again, as we have done earlier to obtain
new amplitudes for each harmonic

// finally playing a random melody to make it less repetitive
(
e.stream = Pbind(
```

```
    \instrument, \partials,
    // frequencies are repeated so we can notice the effect of harmo
nics
    \degree, Pwhite(0, 7),
    \dur, 0.25,
    \stepsPerOctave, 7,
    \octave, 4,
    \pan, Pfunc({0.5.rand2})
).asStream;
)
// we could develop further and re-write the SynthDef with a partia
l argument
// and also change the partials directly from Pbind
```

Figure 17.27
Creating virtual partials for a seven equal-note tuning.

17.7 Tools for Performing and Composing with Alternative Tunings

17.7.1 GUIs

Special tunings deserve interfaces that make them easier to understand and play with. Luckily, SuperCollider has a GUI system that gives much freedom for the design of such user interfaces. As seen earlier with the tonality diamond, buttons can be created to represent the notes from a tuning, not only to trigger them but also to gain a clearer understanding of complex relationships. (See figures 17.28 and 17.29.)

```
(
var w, keys, steps, octaves;

w = Window.new.name = "Custom keyboard: 7 steps per octave";
steps = 7;
octaves = 2;
// seven steps per octave;
a = Array.fill(7, {|i| (1/7)*(i)}) +1;
b = a++(a*2);

c = Synth(\default, [\amp, 0]);

keys = Array.fill(steps*octaves, {|i|
    Button(w, Rect(20+(i*22),20,20,50))
    .states_([
        if(i.mod(steps)==0,{
```

```
                [i.asString, Color.black, Color.red]},{
                      [i.asString, Color.black, Color.yellow]});

    ])
    .action_({arg butt;
         c.set(\freq,b[i]*220, \amp, 0.25)
    });
});

w.front;
)
```

Figure 17.28
Code for a simple GUI keyboard to trigger the seven equal-note temperament.

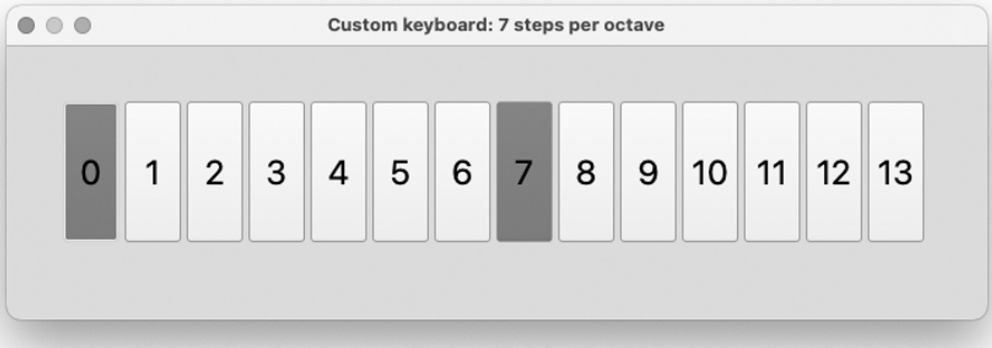


Figure 17.29
Picture of a simple GUI keyboard to trigger the seven equal-note temperament.

GUIS can also help to visualize a mode or anything you think is useful for investigating these tunings. I recommend the ixiViews library (Thor Magnusson, 2006), which can be useful for this type of work.

17.7.2 MIDI

SuperCollider supports MIDI, so it is possible to trigger a MIDI note from a MIDI controller and map it to a special tuning in SuperCollider. Of course, we need to keep track of where the notes are on the keyboard, since we might use more than 12 notes per octave. The computer keyboard can be used to trigger tunings or modes.

17.7.3 Scale and Tuning Classes

The examples in this chapter are just a starting point. Two very useful classes are now in the core of SuperCollider and provide easy access for the exploration of the many

varied and fascinating pitch resources associated with the musics of the world. A `Tuning` object holds a particular tuning system, such as 12-note equal temperament or Partch's 43-note system. A `Scale` is a particular subset of the tuning, such as a major or minor scale. The same scale can be played back with different tunings or different scales applied to different tunings:

```
a = Scale.major   // represents the major scale [0, 2, 4, 5,   7,
 9, 11]
t = Tuning.just;  // just intonation tuning chromatic scale
a.tuning_(t)      // set major scale to use this just tuning
a.semitones       // get (fractional) MIDI note values
a.ratios          // get frequency ratios
```

The `Tuning` and `Scale` classes are very well documented and will be useful to anyone wanting to experiment with alternative systems of pitch, whether they create their own system or want to use a preexisting one:

```
Tuning.directory  //  interesting and famous tunings already   list
ed
Scale.directory   //  interesting and famous scales already listed
```

Creating custom tunings and scales is easy with these classes. Take into account the number of notes that exist in your tuning when creating scales:

```
// custom tuning with 6 unequal values per octave
t = Tuning.new(#[0, 1.3, 2.8, 3.7, 5.2, 8.6], name:   \mynewtunin
g);
// custom scale using all 6 notes from the tuning
b = Scale.new([0, 1, 2, 3, 4, 5], 6, tuning: t);
// play it in ascending pitch order
// note: the seventh degree is an octave higher than degree 0.
Pbind(\scale, b, \degree, Pseq([0,1, 2, 3, 4, 5, 6, 7],inf),   \du
r, 0.25). play;
// play a new scale using only some of the notes from the same tuni
ng c = Scale.new([0, 2, 3, 5], 6, tuning: t);
Pbind(\scale, c, \degree, Pseq([0, 1, 2, 3, 4],inf), \dur,   0.25).
play;
```

17.8 Conclusion

If we consider that Western composers of the past were able to compose such different music with 12 notes, we can deduce that there should be more to alternative tunings than

a few calculations made in the hope of obtaining a sonic piece; a single tuning can already be used in so many different ways. Tuning is a rich area of study. A good starting point would be to get familiar with historical and experimental tunings. With the help of this chapter, you should be then able to write these tunings in SuperCollider code and spend some time listening to them. Because SuperCollider is a wonderful environment for experimentation, you can easily modify code and eventually create your own tunings. One warning for those who grew up under its dominant sway: it can take time for perception to really be free from years of listening to 12-equal-note temperament. As a result, it is important to spend enough time listening to these new tunings in order to appreciate them, recognize what is truly liked, and, eventually, choose what you want to create. There is an infinite space between 1 and 2 within an octave, and thanks to SuperCollider, we can explore more deeply than ever the riches from that territory.

References

Burns, E. M. 1999. "Intervals, Scales, and Tuning." In D. Deutsch, ed., *The Psychology of Music* (2nd ed., pp. 215–264). San Diego: Academic Press.

Carlos, W. 1987. "Tuning: At the Crossroads." *Computer Music Journal,* 11(1): 29–43.

Collins, N. 2003. "Canonic Hill Loss. Microtonal Tempo Canon Generator System after Nancarrow and Jaffe." https://composerprogrammer.com/canonichillloss.html.

Harkins, J. "ddwTemperament." Available in the Quarks.

Magnusson, T. The "ixiViews" library. http://www.ixi-software.net.

McCartney, J. 1996. "SuperCollider Programming Software for the MacIntosh." http://www.audiosynth.com.

Mogini, F. n.d. http://www.fabricemogini.com.

Mogini, F. 2000. "Alternative Tuning Systems." Master's thesis, Lansdown Centre for Electronic Arts, Middlesex University, London.

Partch, H. 1974. *Genesis of a Music,* enlarged ed. New York: Da Capo Press.

Sethares, W. A. 2005. *Tuning, Timbre, Spectrum, Scale,* 2nd ed. Berlin: Springer.

18 Non-Real-Time Synthesis and Object-Oriented Composition

Brian Willkie and Joshua Parmenter

18.1 Introduction

SuperCollider (SC) provides the means for rendering sound works in non-real-time (NRT). SuperCollider's notoriety for real-time performance often overshadows this aspect of the language, but its object-oriented basis, coupled with the extremely powerful signal-processing tools, makes this dimension important to SuperCollider users. Since the sound is rendered offline, the real-time considerations of processor performance are not as crucial (though through good coding practices and the efficient use of UGens, you can still benefit from faster render times). In this chapter we look at object-oriented models that provide elaborate control, allowing the user to create musically expressive code and explore new approaches to sound construction.

18.2 SuperCollider NRT Basics

18.2.1 OSC Commands, Score, and the NRT Server

Before discussing advanced techniques involved with SuperCollider's NRT capabilities and more general compositional tools, a brief description of what SuperCollider needs in order to render a non-real-time sound file is necessary, as well as a brief discussion of problems that arise from using the SuperCollider language beyond its more heavily developed real-time capabilities. SuperCollider's NRT mode relies on binary scores of OSC command bundles.[1] Inside this binary file, only chronologically ordered OSC bundles are allowed. An OSC bundle (in SuperCollider's specialized treatment of OSC messaging) is an array consisting of a time stamp in seconds, and one or more arrays containing OSC commands. For example, `[0.0, [\g_new, 1000, 0, 1]]` is a bundle. Any approach to NRT in SuperCollider needs to format and store the binary OSC messages that the synthesis server (scsynth) requires. Figure 18.1 shows the basic steps to save binary OSC messages for later use offline.

```
/*
This example is adapted and extracted from the Non-Realtime   Synth
esis helpfile itself, accessible from the Main   SuperCollider help
```

```
page.
*/
(
var f, c, d;
// open a file for writing raw OSC data to
f = File("~/test.osc".standardizePath,"w");
// start a sine oscillator at 0.2 seconds.
c = [0.2, [\s_new, \default, 1001, 0, 0]];
// convert the bundle to raw OSC
d = c.asRawOSC;
f.write(d.size); // each bundle is preceded by a 32 bit size.
f.write(d); // write the bundle data.
f.close;
)
```

Figure 18.1
Basic steps to save binary OSC messages for later use offline.

Since most users of the language control events through the server abstraction classes (e.g., `Synth`, `Group`, `Buffer`, and `Bus`), the first hurdle that you may confront in this example is the OSC bundle itself. Working with SuperCollider's NRT capabilities at this level requires an understanding of the OSC communication that occurs between the SuperCollider language and scsynth. Although working directly with raw binary OSC messages can be avoided, if you want to create your own system or classes for use with SuperCollider in NRT, an intimate familiarity with the OSC command reference is necessary. A glance at the source code for the server abstraction classes can also give you some important insights into how the underlying OSC is handled.

A `Score` object gives you a way of managing OSC bundles, provides a basic interface for saving work, and can be used for playing in real time or rendering an event listing offline. Score handles the conversion of OSC bundles into raw OSC and also has the ability to sort events chronologically. The `Score` object, and more specifically its instance variable, `score`, hold an array of bundles. The last time stamp in a Score corresponds to the duration of the output sound file. You can add additional events to a Score, and we recommend that you sort before you save, play, or render a Score, to ensure that it orders the time stamps correctly. Figure 18.2 shows the basic usage of Score through a small example event listing.

```
(
var score;

// A Score, created from a note-list of time-stamped events.
score = Score.new([
```

```
  [0.0,
        // Typically we send a SynthDef to a Real-Time server with
        // .load or .store. For NRT, it's better to include it
        // as part of the score, unless it's a large SynthDef.
        // Refer to the NRT documentation for handling large SynthD
efs.
        ['/d_recv',
              SynthDef(\NRT_beep, {arg freq, dur, amp = 0.1;
                  var half;
                  half = dur * 0.5;
                  Out.ar(0, SinOsc.ar(freq, 0,
                        EnvGen.kr(
                              Env.new([0, amp, 0], [half, half],
 [4, -4])))));
        }).asBytes],
        [\g_new, 1000],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 75.midicps,   \du
r, 0.2]
   ],
   [0.2,
        [\n_free, 1001],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 75.midicps,   \du
r, 0.2]
   ],
   [0.4,
        [\n_free, 1001],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 75.midicps,   \du
r, 0.2]
   ],
   [0.6,
        [\n_free, 1001],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 75.midicps,   \du
r, 0.2]
   ],
   [0.8,
        [\n_free, 1001],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 71.midicps,   \du
r, 0.2]
   ],
   [1.0,
        [\n_free, 1001],
        [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 71.midicps,   \du
r, 0.2]
   ],
```

```
    [1.2,
         [\n_free, 1001],
         [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 82.midicps,   \du
r, 0.2]
    ],
    [1.4,
         [\n_free, 1001],
         [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 82.midicps, \dur,
0.2]
    ],
    [1.6,
         [\n_free, 1001],
         [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 82.midicps,   \du
r, 0.2]
    ],
    [1.8,
         [\n_free, 1001],
         [\s_new, \NRT_beep, 1001, 0, 1000, \freq, 82.midicps,   \du
r, 0.2]
    ],
    [2.0,
         [\n_free, 1001, 1000]
    ],
              [2.00001, [0]]
]);

score.sort;
// Render in Non-Real-Time. Once complete, you can find the audio f
ile
// chptr_1802 in your home directory. recordNRT writes intermediate
files
// in Platform.defaultTempDir, which you can delete when you are do
ne.
score.recordNRT(
    Platform.defaultTempDir ++ "trashme",
    "~/chptr_1802.aiff".standardizePath,
    options: ServerOptions.new.numOutputBusChannels_(1)
);
)
```

Figure 18.2
Example of `Score` usage.

18.2.2 Tips for Score Use

One thing to notice in figure 18.2 is that all of the values are hard-coded (i.e., we've used specific numbers to represent values such as `\freq` and `\dur`). One of the important life lessons of computer programming is "Never hard-code anything!" Suppose that we want all the notes to overlap. Not only do we have to change the duration of every event, but we have to adjust the node IDs as well. Suppose that we want to transpose everything up by a half step. Now suppose that our score is 1,000 events long. Ugh!

We can make large-scale changes easier by storing values in variables and building relationships between them. (See figure 18.3.)

```
(
//In this example, we use higher-level server abstraction classes,
 Group
//and Synth, to handle node IDs. At least as important though, is t
he use
//of variables. Now that we specify relationships rather than speci
fic
//values, we can change the gesture dramatically by changing just o
ne or
//two variables. To transpose everything, we only need to change th
e value
//of ~baseNote. To adjust the duration, we only need to change the
 ~dur
//variable and we can change duration independently of deltaOn.
var score;
var deltaOn = 0.2; //time between the start of one note and   the n
ext
var dur = 0.4; //try changing dur to 0.3, 1.4, 3.4, or whatever   y
ou like
var baseNote = 75; //transpose the entire fragment up or down
var firstPitch = (baseNote + 0).midicps; //alter the interval   bet
ween
var secondPitch = (baseNote - 4).midicps; //first and second   inde
pendently
var thirdPitch = (baseNote + 7).midicps; //of second and third.

score = Score.new([
 [t = 0.0,
   ['/d_recv',
        SynthDef(\NRT_beep, {arg freq, dur, amp = 0.1;
              var half;
              half = dur * 0.5;
```

```
            Out.ar(0, SinOsc.ar(freq, 0,
                EnvGen.kr(
                      Env.new([0, amp, 0], [half, half],   [4, -
4])))));
    }).asBytes],
    (g = Group.basicNew(s)).newMsg,
    // we use environment variables (identified by the preceding ~)
    // since we we don't know how many variables we'll need.
    (~s01 = Synth.basicNew(\NRT_beep, s))
         .newMsg(g, [\freq, firstPitch, \dur, dur], \addToHead)
    ],
    [t + dur,
         ~s01.freeMsg
    ],
    [t = t + deltaOn,
         (~s02 = Synth.basicNew(\NRT_beep, s))
               .newMsg(g, [\freq, firstPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
         ~s02.freeMsg
    ],
    [t = t + deltaOn,
         (~s03 = Synth.basicNew(\NRT_beep, s))
         .newMsg(g, [\freq, firstPitch, \dur, dur], \addToHead)
    ],
    [t + dur,
         ~s03.freeMsg
    ],
    [t = t + deltaOn,
         (~s04 = Synth.basicNew(\NRT_beep, s))
               .newMsg(g, [\freq, firstPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
         ~s04.freeMsg
    ],
    [t = t + deltaOn,
         (~s05 = Synth.basicNew(\NRT_beep, s))
               .newMsg(g, [\freq, secondPitch, \dur, dur], \addToHea
d)
    ],
```

```
    [t + dur,
        ~s05.freeMsg
    ],
    [t = t + deltaOn,
        (~s06 = Synth.basicNew(\NRT_beep, s))
            .newMsg(g, [\freq, secondPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
        ~s06.freeMsg
    ],
    [t = t + deltaOn,
        (~s07 = Synth.basicNew(\NRT_beep, s))
            .newMsg(g, [\freq, thirdPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
        ~s07.freeMsg
    ],
    [t = t + deltaOn,
        (~s08 = Synth.basicNew(\NRT_beep, s))
            .newMsg(g, [\freq, thirdPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
        ~s08.freeMsg
    ],
    [t = t + deltaOn,
        (~s09 = Synth.basicNew(\NRT_beep, s))
            .newMsg(g, [\freq, thirdPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
        ~s09.freeMsg
    ],
    [t = t + deltaOn,
        (~s10 = Synth.basicNew(\NRT_beep, s))
            .newMsg(g, [\freq, thirdPitch, \dur, dur], \addToHea
d)
    ],
    [t + dur,
        ~s10.freeMsg,
```

```
            g.freeMsg
        ],
    ]
    );

    score.sort;
    score.recordNRT(
        Platform.defaultTempDir ++ "trashme",
        "~/chptr_1803.aiff".standardizePath,
        options: ServerOptions.new.numOutputBusChannels_(1)
    );
    )
```

Figure 18.3
Build relationships with variables and functions rather than bard-coding values.

Now, since we have specified the relationships and substituted variables for specific values, when we highlight and evaluate the code, SuperCollider interprets the variables, substituting 622.2539 for every occurrence of `firstPitch`. If we want to transpose the sequence up a fourth, we need to change only the value of `baseNote`. If we want to change the pitch of only the last four notes, we need to change only the value of the variable `thirdPitch`.

In figure 18.3, we also build a temporal relationship with the little bit of code, `t = t + d`, that states, "This event starts at the time of the previous event plus some constant offset." So long as all our events have this relationship, we can substitute that little bit of code for the time stamp of each event (except the first, since it doesn't have a previous event). What's more, we can insert or remove events manually without having to also manually adjust the time stamps of all the subsequent events in our score.

Notice, too, that we used server abstraction classes (`Group`, `Synth`, etc.) to handle the node IDs for us. These classes can provide OSC messages for use with the `Score` class. Synth's `basicNew` method creates a new `Synth` object without sending any data to the server itself. The `newMsg` method returns an OSC message suitable for use with `Score`. These methods allow us to use the server abstraction classes for NRT composition. Although a number of these methods are listed in the Help files, we again suggest that you look through the class definitions for the complete implementation.

If manually adding events to a `Score` seems too labor intensive, then look at figure 18.4, which shows the power of algorithmically creating large `Scores` from within the SuperCollider language itself.

```
(
var score, graingest;
```

```
// seed the randomness
thisThread.randSeed_(123);

score = Score.new;

// envelope times are scaled to 1.
graingest = {arg score, starttime, duration, windur, overlaps,
   freqenv, ampenv, panenv;
   var ratio, curfreq, curamp, curpan, notestart, now = 0.0, note;
   score.add([now, ['/d_rescv',
        SynthDef(\NRT_grain, {arg freq, dur, amp, pan;
              OffsetOut.ar(0, Pan2.ar(
                  SinOsc.ar(freq, 0,
                        EnvGen.ar(Env.sine(dur, amp), doneAction:
 2)),
                  pan)
              );
   }).asBytes]]);

   while({
        ratio = now / duration;
        curfreq = freqenv[ratio];
        curamp = ampenv[ratio];
        curpan = panenv[ratio];
        notestart = now + starttime;
        note = Synth.basicNew(\NRT_grain);
        score.add([notestart,
              note.newMsg(1, [\freq, curfreq,\amp, curamp,
                  \dur, windur, \pan, curpan], \addToHead)]
              );
        // check the current event's endtime against the global end
time
        now = now + (windur / overlaps);
        now < duration;
        });
   };

// call the above function to populate the Score

graingest.value(score, 1.0, 10.0, 100.reciprocal, 1, Env([440,   55
0], [1]), Env([0, 0.2, 0], [0.3, 0.7], [4, -4]), Env([0, 0],
 [1]));
```

```
graingest.value(score, 3.0, 3.0, 130.reciprocal, 2, Env([700,   40
0], [1]), Env([0, 0.2, 0], [0.1, 0.9], [4, -1]), Env([-0.7,   0.7],
[1]));

// create a number of short gestures
10.do({arg i;
   graingest.value(score, 5.0.rrand(10.0), 3.0.rrand(5.0),
        (100 * i).reciprocal, [1, 2, 4].choose,
        Env([1000, 800], [1]),
        Env([0, 0.2, 0], [0.5, 0.5]),
        Env([0.5.rand2, 0.5.rand2], [1]));
   });

// save the endtime to the Score to tell NRT when to stop renderin
g.
// The above gestures won't be more the 16 seconds

score.add([16, [0]]);

// sort the score to ensure events are in the correct order

score.sort;

// render the Score to the users home folder

score.recordNRT(
   Platform.defaultTempDir ++ "trashme",
   "~/chptr_1804.aiff".standardizePath,
   options: ServerOptions.new.numOutputBusChannels_(1)
);

// also save the Score to a file
score.saveToFile("~/test.scd".standardizePath);
)
```

Figure 18.4
Fill a `Score` algorithmically.

While the above examples demonstrate how using variables to define relationships helps us make large-scale changes more easily, they also demonstrate a peculiar feature of `Score`: internally, it stores all of its events as bundles. It doesn't know how to evaluate the variables or the relationships between them that we so carefully created. Thus, every time we make a change, we have to re-create the `Score` before it can reflect

our changes. This approach loses the flexible real-time usage, requiring that we populate a `Score` first, then perform it using the `play` method. Finally, another approach commonly used in the SuperCollider language for NRT composition is the `Pattern` class's `render` and `asScore` methods. This retains some of the flexibility of real-time composing that Pattern supplies and handles the Score production for you. (See chapter 6 for examples.)

18.2.3 Score and Memory Allocation

Special consideration for memory allocation of sound files and buffers (used by UGens such as `BufDelayC` or `FFT`) for offline as well as online rendering of a `Score` is needed. For NRT rendering, you must provide messages for all buffers in the `Score`, and we recommend that you place these OSC messages at the beginning with a time stamp of 0.0. This ensures that any synthesis processes can access the memory when needed. While it is good practice to add OSC bundles at the end of your `Score` to free memory buffers after the NRT process finishes, in reality scsynth takes care of this for you when it exits. However, if you want to use a `Score`'s play method, we suggest that you *not* include the buffers in the `Score` due to the asynchronous nature of buffer allocation. Figures 18.5 and 18.6 show examples for allocating memory within `Score`. In figure 18.5, we place memory allocations in the `Score` because they will take place in NRT and are freed at the end. Figure 18.6 shows the memory allocated in real time. Once the allocation finishes, it is safe to call the play method on the `Score`. When the `Score`'s performance is complete, the memory is released to free up system resources.

```
(
var score, sndbuf, starttime, synth, options;

score = Score.new;

score.add([0,['/d_recv',
   SynthDef(\NRT_playback, {arg buffer, dur, startPos, amp;
   OffsetOut.ar(0, PlayBuf.ar(1, buffer, BufRateScale.kr(buffer),
        startPos: startPos * BufSampleRate.kr(buffer)) *
              EnvGen.ar(
        Env.sine(dur, amp),
        doneAction: 2))
}).asBytes]]);

// create a Buffer object for adding to the Score
sndbuf = Buffer.new;

// for NRT rendering, the buffer messages must be added to the Scor
```

```
e
score.add([0,
   sndbuf.allocReadMsg(
        Platform.resourceDir +/+ "sounds/a11wlk01-44_1.aiff")]);

starttime = 0.0;

// a small function to create a series of small notes based on the
 Buffer
while({
   synth = Synth.basicNew(\NRT_playback);
   score.add([starttime,
        synth.newMsg(s, [\buffer, sndbuf, \dur, 0.1,
              \startPos, 0.0.rrand(1.0), \amp, 0.1])]);
   starttime = starttime + 0.05;
   starttime < 10.0;
   });

// the dummy command. The soundfile will be 11 seconds long
score.add([11, 0]);

score.sort;

// the ServerOptions for rendering the soundfile
options = ServerOptions.new.numOutputBusChannels_(1);

// write the soundfile out to disk
score.recordNRT(
   Platform.defaultTempDir ++ "trashme",
   "~/chptr_1805.aiff".standardizePath,
   options: options
);
)
)
```

Figure 18.5
Memory allocation within `Score` for NRT.

```
(
var score, sndbuf, starttime, synth, options, cond;

SynthDef(\NRT_playback, {arg buffer, dur, startPos, amp;
   OffsetOut.ar(0, PlayBuf.ar(1, buffer, BufRateScale.kr(buffer),
```

```
                startPos: startPos * BufSampleRate.kr(buffer)) *
          EnvGen.ar(
                Env.sine(dur, amp),
                doneAction: 2))
    }).load(s);
score = Score.new;

// set up a Condition to check for when asynchronous events are fin
ished.

cond = Condition.new;

// wrap the code that will run in real-time in a Routine, to allow
 for the Server to sync
Routine.run({
    // load the buffer
    sndbuf = Buffer.read(s, Platform.resourceDir
          +/+ "sounds/a11wlk01-44_1.aiff");

    // pause while the buffer is loaded
    s.sync(cond);

    // fill the Score with notes

    starttime = 0.0;

    while({
          synth = Synth.basicNew(\NRT_playback);
          score.add([starttime,
                synth.newMsg(s, [\buffer, sndbuf, \dur, 0.1,
                      \startPos, 0.0.rrand(1.0), \amp, 0.1])]);
          starttime = starttime + 0.05;
          starttime < 10.0;
          });

    // the last command is NOT needed, since no soundfile is being r
endered score.add([11, 0]);
    score.sort;

    // again, options won't be needed for real time performance opti
ons = ServerOptions.new.numOutputBusChannels_(1);
    score.play;
    // schedule the freeing of the buffer after the Score is done pl
```

```
aying
    SystemClock.sched(11, {sndbuf.free;
         "Buffer resources freed".postln;});
    })
)
```

Figure 18.6
Memory allocation outside of `Score` for real-time usage.

18.2.4 Challenges for NRT Users

Although these approaches offer basic support for NRT work, there are a number of limitations on creating and editing a `Score` or `Pattern`. A number of existing extension libraries help overcome these limitations. In addition, the flexibility of SuperCollider's class library lets you add functionality through the addition of custom class definitions, giving you the ability to expand the possibilities of the language to suit your compositional style and needs.[2] The approach to `Score` creation shown in figures 18.1 and 18.2 reveals some basic problems. Editing compositional details is difficult, and shaping large-scale gestures is even more problematic. The `SynthDef`'s function, with its user-defined parameter list, allows the creation of flexible and modular code. However, since a `Score` stores the OSC commands that set those parameters as specific values, there is no easy way to make large-scale changes to those commands. Those values (usually `Strings`, `Symbols`, and `SimpleNumbers`) have no way of knowing how they relate to the overall gesture. Consequently, if you want to alter the `\freq` parameter contained inside one or more `Score` events, there is simply no good way to do it. Accessing the data is tricky enough, but if you want to change the data from scalar values to time-varying ones, even more problems arise. You have to rewrite `SynthDef` either to include one or more `EnvGens` or to add more OSC messages that create and map the output of a control-rate `Synth` onto the `\freq` parameter of the original.

Figure 18.3 is an excellent example of the problem encountered through the use of the server abstraction classes. These classes are simply not geared toward handling both real-time and NRT concerns, including parameters such as an event's start time and duration. These problems led to the development of the Composer's Tool Kit (Ctk), which is available as a Quark extension library. A basic goal of the system is to take the strengths of the server abstraction classes, while also incorporating concepts of time and methods for setting an event's parameters that look and behave more like the rest of SuperCollider's object-oriented capabilities and language structure.

18.3 Object-Oriented Composition

18.3.1 Composer's Tool Kit

The Composer's Tool Kit (Ctk) is a library of objects that were designed from the beginning to be compatible with both real-time and NRT projects in SuperCollider. Ctk instances are created from the following classes:

`CtkScore`: Stores Ctk events for real-time or NRT performance

`CtkNote`: The basic synthesis object

`CtkNoteObject`, `CtkProtoNotes`, and `CtkSynthDef`: Objects for prototyping `CtkNotes`

`CtkBuffer`: Memory and sound sample management

`CtkControl`: `Control Bus` management and performance

`CtkAudio`: `Audio Bus` management

`CtkEvent`: Encapsulates and controls larger-scale gestures containing other Ctk objects

A `CtkScore` may contain any number of Ctk `Objects`. Like `Score`, `CtkScore` can be played in real time or used for NRT rendering. Any Ctk `Object` can be used in real-time mode (and without the use of `CtkScore`) through use of the play method. Since the system fully supports both approaches, Ctk can be used as an alternative to the server abstraction classes. While the Help files for the Ctk `Objects` (and the examples through the remainder of this chapter) show how the Ctk `Objects` are used, a couple of the system's features should be noted here.

In figure 18.7, a `CtkNoteObject` is used for prototyping `CtkNotes`. After an instance of `CtkNoteObject` is created, the new instance method will create new `CtkNote` objects using the `SynthDef` as a prototype. The resulting `CtkNote` instance automatically creates getter and setter methods for all of the `SynthDef`'s arguments (including those created explicitly through calls to `Control.names` within the `SynthDef`). In play mode, alterations to the control parameters respond in real time to scalars, `Envs`, or `CtkControls`. When `CtkControls` with time-varying signals are used, the system will manage the creation and mapping of additional CtkNotes to the controls.

```
// environment variables are used for real-time examples of Ctk obj
ects
n = CtkNoteObject(
   SynthDef(\NRT_grain, {arg gate = 1, freq, amp;
        var src, env;
        src = SinOsc.ar(freq, 0, amp);
        env = EnvGen.kr(
              Env([0, 1, 0], [1, 1], \sin, 1), gate, doneAction:2);
        OffsetOut.ar(0, src * env);
        })
```

```
    );

// create a new note based on 'n', start to play it in 0.1 seconds
a = n.note(0.1).freq_(440).amp_(0.1).gate_(1).play;
// the release method will set 'gate' to 0.0, and free this node
a.release;

// create another note
a = n.note(0.1).freq_(440).amp_(0.1).play;
// alter the freq argument in real time
a.freq_(550);
// alter the freq with a CtkControl that describes an Env
// CtkControl.env(Env)
a.freq_(CtkControl.env(Env([550, 440, 550], [1, 2], \exp)));
// apply a random control to the amp parameter, with an envelope ap
plied
// to the range. All parameters to the CtkControl can themselves be
// CtkControls. CtkControl.lfo(KRUGen, freq, low, high, phase)
a.amp_(CtkControl.lfo(LFNoise2, 0.5,
    CtkControl.env(Env([0.1, 0.9], [5])), 0.1));
a.amp_(0.1);

// release the note
a.release;
```

Figure 18.7
Example real-time `CtkScore` usage.

For real-time performance, if a `CtkScore` contains `CtkBuffers`, all buffers are allocated before note events begin. In NRT mode, a `CtkBuffer's` messages are placed at the beginning of `CtkScore`, ensuring that memory is properly prepared for notes that need to access it. Figure 18.7 shows some typical real-time uses.

`CtkScores` store the `CtkObjects` themselves, which allows you to easily change the contained objects later. Unlike `Score`, `CtkScore` uses the objects stored inside of it that have a start time and duration to calculate the total duration needed for a rendered sound file. `CtkScore` has the ability to store other `CtkScores`. As a result, you can store layers of events that you can then modify independently of other events. `CtkEvent` is another kind of structure that can encapsulate `CtkObjects`, allowing large-scale gestural shaping. As a result, you are able to manipulate notes, gestures, and even entire pieces on a number of levels. Figure 18.8 shows examples of the power of this approach and how it allows the kind of editing, shaping, and changing of material that we expect in a composition environment. In it, the basic melody initially stored in this instance of `CtkScore` (“Twinkle, Twinkle, Little Star”) is expanded by taking a random

chunk of the melody and reinserting it into itself. This process continues to repeat itself with each new iteration of the expanded `CtkScore` until the desired length is reached. After the creation of the expanded melody, the pitch material is mapped over time into a dynamic pitch range.

```
// melodic expander

var dut, keys, durs, now, score, chunk, expander, rangemap;

thisThread.randSeed_(123);

// a simple note player

dut = CtkSynthDef(\NRT_dut, {arg key, amp, dur;
        Out.ar(0, SinOsc.ar(key.midicps, 0, XLine.kr(amp, 0.00001,
dur)))
        });

// first, make a melody - these will be used as midikeynums
// (easier to alter later)

keys = [72, 72, 79, 79, 81, 81, 79, 77, 77, 76, 76, 74, 74,   72];

// a list of durations

durs = [0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.5,
   0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.5];

// create a var to store 'now' in

now = 0.0;

// create a CtkScore with the above melody

score = CtkScore.new;

keys.do({arg thiskey, inc;
   var thisdur;
   thisdur = durs[inc];
   score.add(dut.note(now,
        thisdur).key_(thiskey).amp_(0.2).dur_(thisdur));
   now = now + thisdur;
   });
```

```
// first, create a function that will return a chunk of the melody
 the
// duration of the chunk sets the starttimes of the notes to a base
of 0.0

chunk = {arg offset = 0;
   var size, start, end, duration = 0, chunk, copies;
   // the size of the current melody - 1 (for array access)
   size = score.notes.size;
   // the beginning of the chunk can come from the beginning of the
melody
   // to the second to last note
   start = 0.rrand(size-1);
   end = start.rrand(size);
   chunk = score.notes[start..end].collect({arg anote;
        var newnote;
        newnote = anote.copy(duration + offset);
        duration = duration + anote.duration;
        newnote;
        });
   [chunk, duration];
   };

// now, create a function that will add those chunks to the score,
 and will keep doing this until the score is at least the desired l
ength. Then check the score size, and truncate to desired size.

expander = {arg len;
   var curchunk, chunkdur, insert, inserttime, insertdur,   cursiz
e, newnotes;
   cursize = score.notes.size;
   while({
        cursize < len
        }, {
        insert = 0.rrand(cursize - 1);
        inserttime = score.notes[insert].starttime;
        insertdur = score.notes[insert].duration;
        #curchunk, chunkdur = chunk.value(inserttime + insertdur);
        score.notes[(insert+1)..(cursize-1)].do({arg me;
              me.setStarttime(me.starttime + chunkdur)});
        score = score.add(curchunk);
```

```
            (score.notes.size > len).if({
                  score.notes.do({arg me, i;
                        (i > (len - 1)).if({score.notes.remove(me)});
                        })
                  });
            cursize = score.notes.size;
            });
      };

// rangemap will place the melodic material within a certain range.
The user passes in an envelope that will describe the center pitch
 in an octave range
rangemap = {arg center;
      score.notes.do({arg me;
            me.key_(me.key.mapIntoRange(12, center[me.starttime]));
            })
      };

// expand it to 100 notes
expander.value(100);

// describe a new range of pitches
rangemap.value(Env([60, 96], [20]));

// finally, play the CtkScore

score.play;
)
```

Figure 18.8
Fill and modify `CtkScore` algorithmically.

Figure 18.8 also demonstrates another important topic for composers of algorithmic music. Since there are random processes involved in these functions, seeding the randomness may be desirable. On the language side of SuperCollider, any thread can have a seed set. In figure 18.8, the `Thread` that the entire SuperCollider interpreter is running is seeded through the `thisThread.randSeed_(seed)` line of code at the beginning of the figure, where "seed" is any integer. When this is set, all random language calls will advance the pseudorandom number generator from this seeded point. You can set the seed again later if you want, and individual threads (such as `Routine`, which is a subclass of `Thread`) can also be seeded. An example of seeding a `Routine` is given in figure 18.9, while also showing another example of the power of `CtkScore`.

```
(
var score, grain, now, thisdur;
var ampmap, double;

grain = CtkNoteObject(
   SynthDef(\NRT_grain, {arg freq, amp, dur, pan = 0;
        var src, env;
        env = EnvGen.ar(
              Env([0, 1, 0], [0.5, 0.5], \sin),
              timeScale: dur, doneAction: 2, levelScale: amp);
        src = SinOsc.ar(freq, 0, env);
        OffsetOut.ar(0, Pan2.ar(src, pan));
        })
   );

score = CtkScore.new;

now = 0;

// create a 3 second granular gesture

while({
   thisdur = 0.05.rrand(0.1);
   score.add(
        grain.note(now, thisdur)
              .freq_(440.rrand(880))
              .amp_(0.05).dur_(thisdur)
              .pan_(0));
   now = now + 0.01;
   now < 3;
  });

// a function to later map the amplitude to a given shape
// envtimes should be scaled to 1
ampmap = {arg aScore, env;
   // scale the env's times by the CtkScore's duration
   env.times = env.times * aScore.endtime;
  aScore.notes.do({arg thisNote;
        var curtime;
        curtime = thisNote.starttime;
        thisNote.amp_(env[curtime]);
        });
  };
```

```
// returns a new copy of the CtkScore with notes
// double an octave higher
double = {arg aScore, shift = 2;
   var thisScore;
   thisScore = aScore.copy;
   thisScore.notes.do({arg thisNote;
        thisNote.freq_(thisNote.freq * shift)
        });
   thisScore;
   };

// a Routine to play the examples
Routine.run({
   var scoreDouble;
   // play the CtkScore;
   score.play;
   score.endtime.wait;
   // remap the amplitudes
   ampmap.value(score, Env([0, 0.2, 0], [0.1, 0.9], [4, -2]));
   1.wait; // pause for a moment
   // play it again!
   score.play;
   score.endtime.wait;
   // add the CtkScore that octaveDouble returns
   scoreDouble = double.value(score, 19.midiratio);
   ampmap.value(scoreDouble, Env([0, 0.25, 0], [0.6, 0.4],   [4, -
2]));
   score.add(scoreDouble);
   1.wait;
   score.play;
   score.endtime.wait;
   // don't like the second version? remove double
   score.ctkscores.remove(scoreDouble);
   ampmap.value(score, Env([0.15, 0.05], [1]));
   1.wait;
   score.play;
   }).randSeed_(123)
)
```

Figure 18.9
Build a granular gesture with Ctk for offline rendering algorithmically.

The example has been built from a basic gesture, along with some functions that modify the gesture in a couple of simple ways. A `Routine` at the end of the code plays, then alters, the gesture in steps for you to audition. The example shows how you are able to modify `CtkNotes` that are stored inside a `CtkScore`. In this case, a couple of functions treat the `CtkScore` and its contents as a single gesture. First, we create a simple granular gesture that lasts 3 seconds. Frequency is random, and the amplitude is flat. The `ampmap` function will iterate over the `CtkScore`'s notes and alter each `CtkNote`'s `amp` parameter according to an `Env` object. The modification is made in place, and once it is finished, we can play the `CtkScore` again to hear the changes. The second function, `double`, creates a copy of a `CtkScore` and scales the `freq` parameter of every note by a value. Then this second `CtkScore's` amplitude curve is remapped to new values. This copied and altered `CtkScore` is added to the first one, and then the new version is performed. In the final step, we remove the second `CtkScore` and remap all of the `CtkNote`'s amplitudes to the original level. While it is certainly possible to create and shape multiple layers of a piece in this fashion, the ideas of shaping the amplitudes of the event's objects or transposing a frequency over an entire `CtkScore` are far from specialized cases. Once we find ourselves treating an object and everything it contains in a unified way, then why not create objects that will do all of this for us?

18.3.2 Encapsulation

One of the fundamental elements of music, regardless of style, is varied repetition. The relationship between a subclass and its superclass is precisely this, a varied repetition. More than just a metaphor, object-oriented programming (OOP) codifies a set of general programming practices that can aid the computer musician in creating music. One of the basic elements of OOP is encapsulation (i.e., grouping related things together). In the score in figure 18.10, we have a group containing a simple oscillator that is controlled by a low-frequency oscillator (LFO) and an amplitude envelope.

```
(
// CtkScore.write will call SynthDef.asBytes for us, so we can
// define our CtkSynthDef outside of the score, regardless of
// whether we use it in realtime or non-realtime.
var sinosc = CtkSynthDef.new(\NRT_sinosc,
   {arg outbus = 0, freq = 622.254, phase = 0, amp = 1, offSet = 0;
   Out.ar(outbus, SinOsc.ar(freq, phase, amp, offSet));
   }
);

var score;
var baseNote = 75;
```

```
var slopeTime = 0.25;
var curve = \sine;
var firstPitch = (baseNote + 0).midicps;
var firstStart = 0.0;
var firstDur = 5.0;
var firstAttackTime = slopeTime * 0.5;
var firstDecayTime = slopeTime - firstAttackTime;
var firstVibDepth = 0.21;
var firstVibRate = 2.3;
var firstPeakAmp = 0.25;
var firstDecayAmp = 0.01;
var secondPitch = (baseNote - 4).midicps;
var secondStart = 2.2;
var secondDur = 4.0;
var secondAttackTime = slopeTime * 0.5;
var secondDecayTime = slopeTime - secondAttackTime;
var secondVibDepth = 0.15;
var secondVibRate = 1.7;
var secondPeakAmp = 0.25;
var secondDecayAmp = 0.01;
var thirdPitch = (baseNote + 7).midicps;
var thirdStart = 3.1;
var thirdDur = 3.75;
var thirdAttackTime = slopeTime * 0.5;
var thirdDecayTime = slopeTime - thirdAttackTime;
var thirdVibDepth = 0.21;
var thirdVibRate = 4;
var thirdPeakAmp = 0.2;
var thirdDecayAmp = 0.25;

score = CtkScore.new(
   ~firstGroup = CtkGroup.new(firstStart, firstDur, server: s),
   sinosc.note(firstStart, firstDur, \tail, ~firstGroup, server: s)
        .freq_(CtkControl.lfo(SinOsc, firstVibRate,
              (firstPitch - ((firstPitch / (firstPitch.log2))
                    * (firstVibDepth * (1/3)))),
              (firstPitch + ((firstPitch / (firstPitch.log2))
                    * (firstVibDepth * (2/3)))),
              duration: firstDur,
              addAction: \head,
```

```
                target: ~firstGroup,
                server: s))
        .amp_(CtkControl.env(
                Env.new(
                        [0, firstPeakAmp, firstDecayAmp, 0],
                        [firstAttackTime,
                            firstDur - (firstAttackTime +  firstDecayT
ime),
                            firstDecayTime],
                        curve),
                addAction: \head,
                target: ~firstGroup,
                server: s)),
    ~secondGroup = CtkGroup.new(secondStart, secondDur, server: s),
    sinosc.note(secondStart, secondDur, \tail, ~secondGroup,  serve
r: s)
        .freq_(CtkControl.lfo(SinOsc, secondVibRate,
                (secondPitch - ((secondPitch / (secondPitch.log2))
                        * (secondVibDepth * (1/3)))),
                (secondPitch + ((secondPitch / (secondPitch.log2))
                        * (secondVibDepth * (2/3)))),
                duration: secondDur,
                addAction: \head,
                target: ~secondGroup,
                server: s))
        .amp_(CtkControl.env(
                Env.new(
                        [0, secondPeakAmp, secondDecayAmp, 0],
                        [secondAttackTime,
                              secondDur - (secondAttackTime +  secondD
ecayTime),
                        secondDecayTime],
                        curve),
                addAction: \head,
                target: ~secondGroup,
                server: s)),
    ~thirdGroup = CtkGroup.new(thirdStart, thirdDur, server: s),
    sinosc.note(thirdStart, thirdDur, \tail, ~thirdGroup, server: s)
        .freq_(CtkControl.lfo(SinOsc, thirdVibRate,
                (thirdPitch - ((thirdPitch / (thirdPitch.log2))
                        * (thirdVibDepth * (1/3)))),
                (thirdPitch + ((thirdPitch / (thirdPitch.log2))
                        * (thirdVibDepth * (2/3)))),
                duration: thirdDur,
```

```
                addAction: \head,
                target: ~thirdGroup,
                server: s))
          .amp_(CtkControl.env(
                Env.new(
                      [0, thirdPeakAmp, thirdDecayAmp, 0],
                      [thirdAttackTime,
                            thirdDur - (thirdAttackTime +   thirdDeca
yTime),
                thirdDecayTime],
                curve),
          addAction: \head,
          target: ~thirdGroup,
          server: s))
);

score.write(
   oscFilePath: Platform.defaultTempDir ++ "trashme",
   path: "~/chptr_1810.aiff".standardizePath,
   options: ServerOptions.new.numOutputBusChannels_(1)
);
)
```

Figure 18.10
Example of non-real-time Ctk usage.

These four objects are always together. Every time we have the oscillator, we have a group, an LFO, and an envelope.

Since these objects are related, we can group them together and give them a name, such as `VSO` (`VibratoSinOsc`). Then, whenever we want an oscillator, an LFO, and an envelope, we just have to type something like

```
VSO.new;
```

Likewise, there are a number of parameters in common for every use of our combination `group-oscillator-lfo-envelope`: pitch, start, duration, vibrato depth, vibrato rate, peak amplitude, and decay amplitude. One clue that we can group these variables together into one object is that all their names have something in common: *first*Pitch, *first*Start, or *first*Dur. Finally, any time you see code that is copied and pasted (or copied and pasted with slight modifications, such as changing a variable name), that's a strong indication that such code can be grouped together, at least in a function, if not a class. For example, the bit of code `(firstPitch-((firstPitch/`

`(firstPitch.log2)) * (firstVibDepth * (1/3)))` occurs with every instance of our `oscillator-lfo-envelope` grouping, with changes only to the variable name. We can rewrite it as a function as follows:

```
f = {arg pitch, vibDepth; pitch—((pitch/(pitch.log2)) *   (vibDepth
* (1/3)))};
```

Then, everywhere we have that bit of code, we can replace it with a call to our function:

```
CtkControl.lfo(SinOSsc, firstVibRate, f.value(firstPitch,   firstVi
bDepth));
```

and similarly for the LFO's high argument.

When building a class (actually, when writing any code), it's good to have a clear idea of your goal. In this case, we want a `group-oscillator-lfo-envelope` object that we can add to a `CtkScore`:

```
VSO {
  var < score, group, oscil, freqCntl, ampCntl;
}
```

We'll need a constructor with all the parameters that we want to allow the user to set:

```
*new {arg start = 0.0, dur = nil, freq = 622.254, ampPeakLevel =
 0.707, ampDecayLevel = 0.001, vibDepth = 0.21, vibRate = 3,   addA
ction = 0, target = 1, server;
   ^super.new.initVSO(start, dur, freq, ampPeakLevel,   ampDecayLev
el, vibDepth, vibRate, addAction, target,   server);
}
```

The constructor allocates memory for our objects (`super.new`) and then calls our initialization method, `initVSO`. Note the naming convention of our initialization method:

```
init<ClassName>
```

We strongly encourage this convention. For more details, see the "New Instance Creation" section of the WritingClasses Help file included in the SuperCollider distribution.

The initialization method is where most of the work is done, but before we dive into it, we have a couple of additional opportunities to encapsulate: the frequency and amplitude controls:

```
.freq_(CtkControl.lfo(SinOsc, firstVibRate,
   (firstPitch-((firstPitch/(firstPitch.log2)) * (firstVibDepth   *
(1/3)))),
   (firstPitch-((firstPitch/(firstPitch.log2)) * (firstVibDepth   *
(2/3)))),
   addAction: \head, target: _firstGroup, server: s))
```

Notice that every time we create a `CtkControl` for frequency, we have this little bit of code, and it occurs only when we create that control. Those facts make it a good candidate for encapsulation. We can group the `CtkControl`, the various data that it requires (e.g., pitch, rate, depth), and the code that manipulates those data (the functions that give us our low and high values for our LFO) and call it something like `VSO_Vib`. (See figure 18.11.)

```
VSO_Vib {

   var <pitch, <depth, <rate, <control;

   *new {arg start = 0.0, dur = nil, freq = 1, vibDepth = 0.21,
        vibRate = 1, addAction = 0, target = 1, server;
        ^super.new.initVSO_Vib(start, dur, freq, vibDepth,
             vibRate, addAction, target, server);
   }

   initVSO_Vib {arg start, dur, freq, vibDepth,
        vibRate, add = 0, tgt = 1, server;
        server = server ?? {Server.default};
        pitch = freq;
        depth = vibDepth;
        rate = vibRate;
        control = CtkControl.lfo(SinOsc, rate, this.getLowerValue,
             this.getUpperValue, 0, start, dur, add, tgt, server:
server);

   }

   getLowerValue {
```

```
            ^(pitch - ((pitch / (pitch.log2)) * (depth * (1/3))));
        }

        getUpperValue {
            ^(pitch + ((pitch / (pitch.log2)) * (depth * (2/3))));
        }
    }
```

Figure 18.11
VSO_Vib class definition.

Like our VSO object, it has a constructor that allocates memory and calls our initialization method (`initVSO_Vib`) that does all the work of creating and initializing our `CtkControl.lfo`. Additionally, we've encapsulated our functions that determine the lower and upper limits of our LFO into the methods `getLowerLimit` and `getUpperLimit`, respectively. We can apply similar criteria to our amplitude control. (See figure 18.12.)

```
VSO_ADR {

    var <control, <attackDur, <releaseDur, <totalDur;

    *new {arg start = 0.0, dur = nil, peak = 0.707, decay = 0.01,
            attackDur = 0.125, releaseDur = 0.125, addAction = 0,
            target = 1, server;
            ^super.new.initVSO_ADR(start, dur, peak, decay,   attackDu
r,
                releaseDur, addAction, target, server);
    }

    initVSO_ADR {arg start = 0.0, dur = nil, peak = 0.707,
            decay = 0.01, aDur = 0.125, rDur = 0.125, addAction = 0,
            target = 1, server;
            server = server ?? {Server.default};
            attackDur = aDur ?? 0.0;
            releaseDur = rDur ?? 0.0;
            totalDur = dur.isNil.if({(attackDur + releaseDur)},
                  (dur < (attackDur + releaseDur)).if(
                        {(attackDur + releaseDur)},
                        {dur}));
            control = CtkControl.env(Env.new([0, peak, decay, 0],
                  [attackDur, this.decayDur, releaseDur], \sine),
```

```
                start, addAction, target, server: server,   doneActio
n: 0);
    }

    decayDur {
         ^(totalDur - (attackDur + releaseDur));
    }

}
```

Figure 18.12
`VSO_ADR` class definition.

But doesn't all this belong to our VSO object? Well, yes, but perhaps not exclusively. We might want to apply vibrato to things other than an oscillator, such as the center frequency of a bandpass filter or another LFO. If we lock this code inside our VSO class, we'll have to rewrite it every time we want to apply vibrato to something else. Later, if we decide that we want to change the way that we implement vibrato (say that we want to add jitter to it or use a different library), we'll have to update every class that implements vibrato. Conversely, if we separate the vibrato out, we can still use it in our VSO object and it will be available for use to any other class. If we decide to change our implementation of vibrato, we have to change it in only one place.

This raises another important life lesson in computer programming: "Write modular code!" Modular code isolates functionality into the smallest bit of code that has a practical use. By grouping together things that are closely related, and only things that are closely related, encapsulation promotes modular design. Likewise, practicing modular design leads to grouping together only the things that are really related to each other.

Now that we have our `VSO_Vib` and `VSO_ADR` classes, we can use them in our `initVSO` method. (See figure 18.13.)

```
initVSO {arg start = 0.0, dur = nil, freq = 622.254,
    ampPeakLevel = 0.707, ampDecayLevel = 0.01, vibDepth =   0.21,
    vibRate = 3, addAction = 0, target = 1, server;
    server = server ?? {Server.default};
    group = CtkGroup.new(start, dur, addAction: addAction,
         target: target, server: server);
    freqCntl = VSO_Vib.new(start, dur, freq, vibDepth, vibRate,
         \head, group, server);
    ampCntl = VSO_ADR.new(start, dur, ampPeakLevel, ampDecayLevel,
         addAction: \head, target: group, server: server);
```

```
    oscil = sinoscdef.note(start, dur, \tail, group, server)
         .freq_(freqCntl.control).amp_(ampCntl.control);
    score = CtkScore.new(group, oscil);
    }
```

Figure 18.13
VSO initialization method.

Like our initialization methods for `VSO_Vib` and `VSO_ADR`, `initVSO` simply instantiates the objects for our instance variables `score`, `group`, `oscil`, `freqCntl`, and `ampCntl`. One interesting difference between VSO and `VSO_Vib` and `VSO_ADR` is its class variable `sinoscdef` and class method `*initClass`.[3] Here, `sinoscdef` is the `CtkSynthDef` that describes our oscillator:

```
classvar <sinoscdef;
*initClass {
              StartUp.add({
   sinoscdef.isNil.if({
        sinoscdef = CtkSynthDef.new(\NRT_sinosc,
              {arg outbus = 0, freq = 622.254, phase = 0, amp =
 1, offset = 0;
              Out.ar(outbus, SinOsc.ar(freq, phase, amp, offset));
              })
        })
});
}
```

Since we don't change this for any instance of VSO, we can define it once at the class level, so it is available for all VSO objects. It is even available to other objects without creating an instance of VSO:

```
a = VSO.sinoscdef;
```

Note that `StartUp.add` instructs SC to compile its library before compiling this code, ensuring the `SynthDef` class exists before we attempt to use it (via `CtkSynthDef`). So now we have our completed class definition. (See figure 18.14.)

```
VSO {

    classvar <sinoscdef;
    var <score, group, oscil, freqCntl, <ampCntl;
```

```
   *new {arg start = 0.0, dur = nil, freq = 622.254,
        ampPeakLevel = 0.707, ampDecayLevel = 0.01, vibDepth =   0.
21,
        vibRate = 3, addAction = 0, target = 1, server;
        ^super.new.initVSO(start, dur, freq, ampPeakLevel,   ampDec
ayLevel,
              vibDepth, vibRate, addAction, target, server);
   }

   *initClass {
        StartUp.add({
              sinoscdef.isNil.if({
                   sinoscdef = CtkSynthDef.new(\NRT_sinosc,
                        {arg outbus = 0, freq = 622.254, phase = 0,
amp = 1,
                                  offSet = 0;
                                  Out.ar(outbus,
                                      SinOsc.ar(freq, phase, amp, offSe
t)
                                  );
                   })
              })
        })
   }

   initVSO {arg start = 0.0, dur = nil, freq = 622.254,
        ampPeakLevel = 0.707, ampDecayLevel = 0.01, vibDepth =   0.
21,
        vibRate = 3, addAction = 0, target = 1, server;
        server = server ?? {Server.default};
        group = CtkGroup.new(start, dur, addAction: addAction,
              target: target, server: server);
        freqCntl = VSO_Vib.new(start, dur, freq, vibDepth, vibRate,
              \head, group, server);
        ampCntl = VSO_ADR.new(start, dur, ampPeakLevel,   ampDecayL
evel,
              addAction: \head, target: group, server: server);
        oscil = sinoscdef.note(start, dur, \tail, group, server)
              .freq_(freqCntl.control).amp_(ampCntl.control);
        score = CtkScore.new(group, oscil);
        }
```

```
}
```

Figure 18.14
Complete VSO class definition.

"That's a lot of code!" you might say. Well, it is more than our initial short score, but let's take a look at that score after substituting our new class. (See figure 18.15.) Notice that we've taken advantage of the fact that we can nest instances of `CtkScore` within another `CtkScore`. We've reduced the number of lines in our score by nine-tenths, from 30 in the first example (figure 18.10) to 3 lines in our last example (figure 18.15). Now imagine that you want to create 100 vibrato-oscillators. What if you want to add LFOs to modulate the phase and frequency as well? But the benefits don't stop with the need for less typing. Look at how much easier it is to read our last example than the first. Only the necessary details are present in the last example: that we have three objects that produce a vibrating sine wave. We don't need to know the specifics of `CtkGroup`, `CtkSynthDef`, and `CtkControl`; too much detail simply clutters up the score.

```
(
var score;
var baseNote = 75;
var firstPitch = (baseNote + 0).midicps;
var secondPitch = (baseNote - 4).midicps;
var thirdPitch = (baseNote + 7).midicps;

score = CtkScore.new(
    (VSO.new(0.0, 5.0, firstPitch, 0.25, 0.01, 0.21, 2.3)).score,
    (VSO.new(2.2, 4.0, secondPitch, 0.25, 0.01, 0.15, 1.7)).score,
    (VSO.new(3.1, 3.75, thirdPitch, 0.15, 0.3, 0.21, 4)).score
);

score.write(
    oscFilePath: Platform.defaultTempDir ++ "trashme",
    path: "~/chptr_1815.aiff".standardizePath,
    options: ServerOptions.new.numOutputBusChannels_(1)
);
)
```

Figure 18.15
Example of `CtkScore` from figure 18.10 with VSO substitution.

Encapsulation is a very simple idea: group together related objects and treat then like one thing. Yet it is a very powerful way to organize code that you use repeatedly and to make your code more readable.

18.3.3 Polymorphism

As an exercise, we can encapsulate some information that is common to Ctk objects—start time and duration:

```
NRT_TimeFrame {
   var <>starttime, <>duration;
   *new {arg starttime, duration;
        ^super.newCopyArgs(starttime, duration);
   }
}
```

We could also add a common Ctk message—end time:

```
endtime {
   ^starttime + duration;
}
```

Notice that the relationship between the start of the first note and the start of the second note is clearer. It's certainly true that we could simply specify a start time of 2.4 for the second note. After all, it's not that hard to add 0.0 and 2.4, but that misses the point. It's better to identify the relationship and let the computer fill in the details. The fewer details you provide explicitly, the less work it takes to make changes later.

While we expect the user to fill in the start time and duration with a `SimpleNumber` (i.e., an `Integer` or a `Float`), it might be interesting to support other types of objects, such as `Functions`:

```
d = [2.4, 1.7];
a = NRT_TimeFrame.new(0.0, 11);
b = NRT_TimeFrame.new({a.starttime + d.at(0)}, {a.endtime-(a.   sta
rttime + d.at(0))});
c = NRT_TimeFrame.new({b.starttime + d.at(1)}, {a.endtime-(b.   sta
rttime + d.at(1))});
```

However, we wouldn't want to add a start time function to a duration function in our end time method.[4] One way to handle this would be to check the type for start time and duration before we return a value:

```
endtime {
    var start, dur;
    if((starttime.isKindOf(Function)), {
         start = starttime.value;
    }, {
         start = starttime;
    });
    if((duration.isKindOf(Function)), {
         dur = duration.value;
    }, {
         dur = duration;
    });
         ^start + dur;
}
```

However, there is a better way. We can take advantage of the fact that `Object` implements a `value` method. Consequently, everything in SuperCollider, regardless of type, responds to value.[5] This is an example of polymorphism. When different objects implement the same method, we can write code that doesn't depend on the type of object at hand. It may mean something entirely different for a `SimpleNumber` to respond to a value message than for a `Function` to do so, but because they both respond to it, we don't have to overly concern ourselves with those details. We know that we can call value on anything that is passed in, so we can discard all that type checking and just get to work.

```
endtime {
    ^starttime.value + duration.value;
}
```

But end time is not the only thing to use start time and duration. The user is likely to access them, too, so in this case we can build our own getter methods.

```
starttime {
    ^starttime.value;
}
duration {
    ^duration.value;
}
```

Now, within the end time method we can call the getter methods rather than accessing the instance variables directly. We get the same results the user would get for start time

and duration; and what's more, we protect the implementation of end time from any changes we make to the getter methods.

Another consideration is that the user might want to leave the start time and/or the duration empty (i.e., `nil`). nil responds to value, just like everything else in SuperCollider, but if we try to add nil to something else, that will throw an error. So we probably want a little bit of error checking to make sure that we don't try to add nil when we want to determine the end time. Figure 18.16 shows the complete class definition.

```
NRT_TimeFrame {

    var >starttime, >duration;

    *new {arg starttime, duration;
         ^super.newCopyArgs(starttime, duration);
    }

    starttime {
         ^starttime.value;
    }

    duration {
         ^duration.value;
    }

    endtime {
         ^(this.starttime != nil).if({
              (this.duration != nil).if({
                   //call the getter methods rather than accessing
                   // the variables directly
                   this.starttime + this.duration;
              }, {nil})
         }, {nil});
    }

}
```

Figure 18.16
Complete `NRT_TimeFrame` class definition.

Now we can clean up that timing code we had in figure 18.15. (See figure 18.17.)

```
(
var score;
var baseNote = 75;
var firstPitch = (baseNote + 0).midicps;
var secondPitch = (baseNote - 4).midicps;
var thirdPitch = (baseNote + 7).midicps;

d = [2.4, 1.7];
a = NRT_TimeFrame.new(0.0, 11);
b = NRT_TimeFrame.new(a.starttime + d.at(0),
   a.endtime - (a.starttime + d.at(0)));
c = NRT_TimeFrame.new(b.starttime + d.at(1),
   b.endtime - (b.starttime + d.at(1)));

score = CtkScore.new(
   (VSO.new(a.starttime, a.duration, firstPitch,
        0.25, 0.01, 0.21, 2.3)).score,
   (VSO.new(b.starttime, b.duration, secondPitch,
        0.25, 0.01, 0.15, 1.7)).score,
   (VSO.new(c.starttime, c.duration, thirdPitch,
        0.15, 0.3, 0.21, 4)).score
);

score.write(
   oscFilePath: Platform.defaultTempDir ++ "trashme",
   path: "~/chptr_1816.aiff".standardizePath,
   options: ServerOptions.new.numOutputBusChannels_(1)
);
)
```

Figure 18.17
Example of `CtkScore` from figure 18.15 using `NRT_TimeFrame`.

Polymorphism can be defined simplistically as differing objects implementing a method of a given name. But this glosses over many of the profound and extensive impacts it has on computer programming. The best we can do here is to offer some introductory examples. As you spend more time with SuperCollider and programming for computer music, it will be worth your while to seek out more detailed examples and how to apply them to your work to make your life easier. Some suggested readings are listed at the end of this chapter.

18.4 A Customizable Object-Oriented Approach to Music Composition

SuperCollider is a versatile language supporting a wide variety of compositional approaches, from live coding, interactive installations, and other real-time techniques to computationally expensive algorithmic pieces and old-fashioned, fixed studio works more suited to NRT techniques. Its support for object-oriented programming sets it apart from many of its predecessors and is the major reason why most people find it so appealing. All the same, most of us still find it challenging to use. In this chapter, we've looked at ways to leverage OOP concepts and third-party libraries such as the Composer's Tool Kit to make our lives easier.

Object-oriented programming has been developed over several decades. Many books and college courses are devoted to the topic, and some of the earliest examples of its application to music can be found in the work of Henry Lieberman, Stephen Pope, and many others dating back to the late 1980s and early 1990s. As noted earlier, the best we can do here is to offer some introductory examples. As you spend more time with SuperCollider and programming for computer music, it will be worth your while to seek out more detailed examples and consider how to apply them to make your work easier. Some suggested readings include *Object-Oriented Design Heuristics* by Arthur J. Riel, *OOP Demystified* by James Keogh and Mario Giannini, and *An Introduction to Object-Oriented Programming* by Timothy Budd, the author who gave us *A Little Smalltalk,* which is recommended in the SuperCollider Help file StreamsPatternsEvents1.html.

Likewise, Ctk is a sizable library, rich in functionality. We've only scratched the surface here. Check out the additional NRT materials on the supplemental website for some more examples.

Although we have focused on NRT approaches such as algorithmic composition and fixed studio works, the figures we've presented are not meant as a prescription for any particular compositional method, nor are the tools and techniques we've described limited to NRT usage. Rather, we present these figures as opportunities to explore ways to codify the relationships inherent in music. It is of course up to you to define the relationships that are significant to you, the programming techniques that best represent those relationships, and the tools that are most useful to you. Good luck!

Notes

1. It is important to be familiar with the specialized implementation of OSC that SuperCollider uses. There are some examples in this chapter that may provide a basic introduction to how OSC works in SuperCollider, but the ServerCommandReference Help file that is linked from the main SuperCollider Help page gives a more complete overview.

2. `Score` started out as an extension put together by Josh Parmenter for reading event listings from the CommonMusic library extension to Lisp. This was later greatly improved by Scott Wilson, Julian Rohrhuber, and James McCartney before becoming a part of the standard distribution and is one of the main tools for NRT work.
3. **initClass* is called by *Class:initClassTree* in *Process:startup* for everything that is inherited from `Object`.
4. Actually, it is possible to add `Functions`. See the SuperCollider Help documentation on `Function` for more information.
5. This is because everything in SuperCollider is a subclass of `Object`.

References

Budd, T. 2001. *ObjectOriented Programming*. Reading, MA: Addison-Wesley.

Budd, T. 1987. *A Little Smalltalk*. Reading, MA: Addison-Wesley.

Keogh, J., and M. Giannini. 2004. *OOP Demystified*. McGraw-Hill Osborne Media.

Pope, S. T., ed. 1991. *The Well-Tempered Object: Musical Applications of ObjectOriented Software Technology*. Cambridge, MA: MIT Press.

Riel, A. J. 1996. *ObjectOriented Design Heuristics*. Reading, MA: Addison-Wesley Professional.

19 Stochastic and Deterministic Algorithms for Sound Synthesis and Composition

Sergio Luque and Daniel Mayer

Any algorithm can be the source material for sound synthesis and algorithmic composition, whether it has an existing link to musical acoustics or not. In this chapter, we explore some of the more esoteric approaches, inspired by procedures that wouldn't typically appear in an acoustics or digital signal processing textbook, so-called "nonstandard synthesis" (Holtzman 1978). These algorithms can be entirely deterministic, such as number sequences, fractal constructions, or chaotic dynamics, or involve different degrees of randomness where the next step may depend on a roll of the dice, such as a random walk. Composers can explore a whole universe of sound and structure generation, favoring exploration over more traditionally oriented purposes (see also chapter 16, on microsound).

Since the appearance of computers with digital to analog converters, composers have been interested in synthesizing sound and composing sound structures directly by manipulating amplitude values (sample values). In this approach, amplitude values are not based on acoustic models and are not produced by traditional unit generators. Instead, they are generated through techniques coming from instrumental music composition, arithmetic and logical operations, and other esoteric methods. These practices have often been referred to as "nonstandard synthesis," and, in some cases, they have been described as "guerrilla sound synthesis" (Berg 2009), "instruction synthesis" (Roads 1996), and "errant synthesis" (Collins 2008). SAWDUST by Herbert Brün, SSP by Gottfried Michael Koenig and Dynamic Stochastic Synthesis by Iannis Xenakis are three pioneering nonstandard synthesis strategies that appeared in parallel during the 1970s. They share the following goals: to unify the macrostructure and microstructure of compositions, to use synthesis techniques idiomatic to computers, and to open an experimental field in sound synthesis. Taking the four-class taxonomy of Julius O. Smith III (1991), the presented synthesis methods fall into the categories of *abstract algorithms* (19.1, 19.2, 19.3, 19.4, and 19.5) and *processed recordings* (19.4 and 19.5), omitting spectral and physical models.

Concerning the demanding question of how to connect the development of sound, gestures, and overall form (which might be micro-, meso-, and macro-level), there

cannot be a single recommendation. However, we present some use cases where a procedure can apply at all levels (19.1 and 19.2) or the meso- and macro-level structure directly results from a synthesis setup (19.3). Such generic frugality can be an attractive artistic strategy. Frugality has also been a guiding theme for us in terms of implementation; all examples use plain SuperCollider without extensions, and are relatively short.

Random walks and tendency masks are two stochastic processes that can be valuable for algorithmic composition and sound synthesis. They offer two ways of harnessing random numbers to model behaviors and shapes that unfold over time. Random walks can create continuity and variation in an organic and unpredictable manner, while tendency masks can give direction and tension. Sections 19.1 and 19.2 show how to implement both techniques in SuperCollider to generate pitches, durations, synthesis parameters, and values for nonstandard synthesis. To get these processes to work successfully is often not so much a matter of making code more complex as it is of fine-tuning them by listening carefully to the musical or sonic consequences of changes in their parameters.

19.1 Random Walks

A *random walk* is a stochastic process that describes a path consisting of a sequence of random steps. In 1827, botanist Robert Brown observed through his microscope particles of pollen suspended in water and noticed that they moved continuously and erratically. In 1905, Albert Einstein explained that this movement was caused by the constant random bombardment of the molecules of water. The movement of the particles is known as *Brownian motion*, which can be thought of as a random walk.

To implement a random walk, we start at the initial position of the path and calculate each new position by adding a random step to the previous one. To control the successive generation of steps, we need to select a probability distribution and its parameters to determine the behavior of the random walk and the sizes of its steps.

Here is how to create and plot a symmetric random walk (−1 and 1 have equal probabilities) of 500 steps (figures 19.1 and 19.2).

```
(
var walk = [0]; // initial position
// 500 new positions are calculated by adding steps of −1 or // 1 t
o the previous position
500.do({walk = walk.add(walk.last + [−1, 1].choose)});
walk.plot(discrete: true)
)
```

Figure 19.1
Code for a 500-step symmetric random walk.

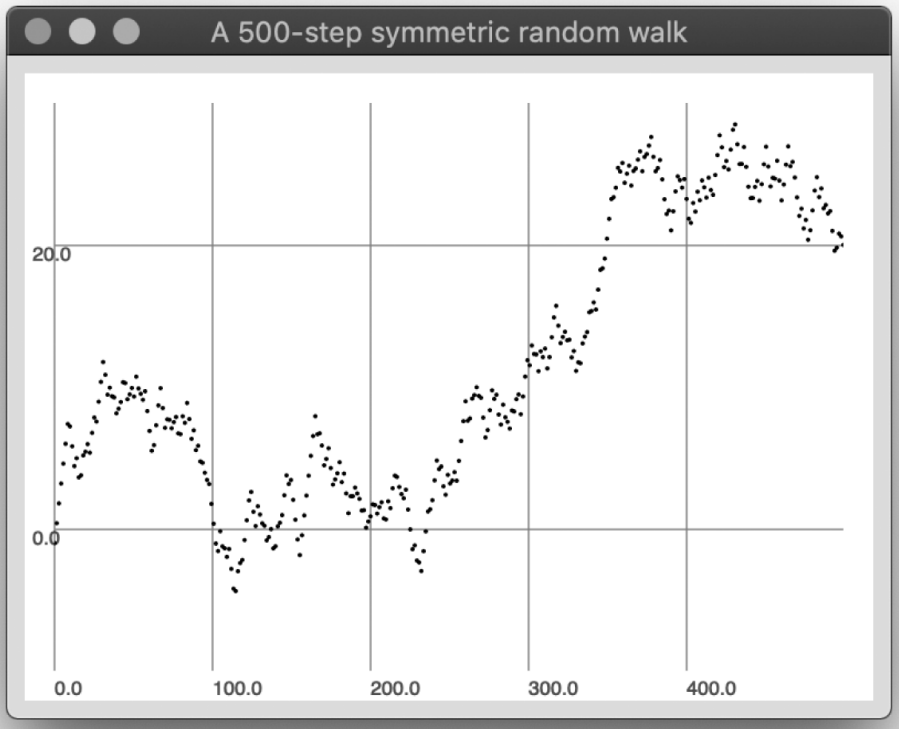


Figure 19.2
Picture of a 500-step symmetric random walk.

We can create one hundred 500-step symmetric random walks and plot them to compare their paths (figures 19.3 and 19.4).

```
(

({
 var walk = [0];
 500.do({walk = walk.add(walk.last + [-1, 1].choose)});
 walk
}!100).plot(discrete: true).plotColor_(({Color.rand}!100)).   super
pose_(true)
)
```

Figure 19.3
Code for one hundred 500-step symmetric random walks.

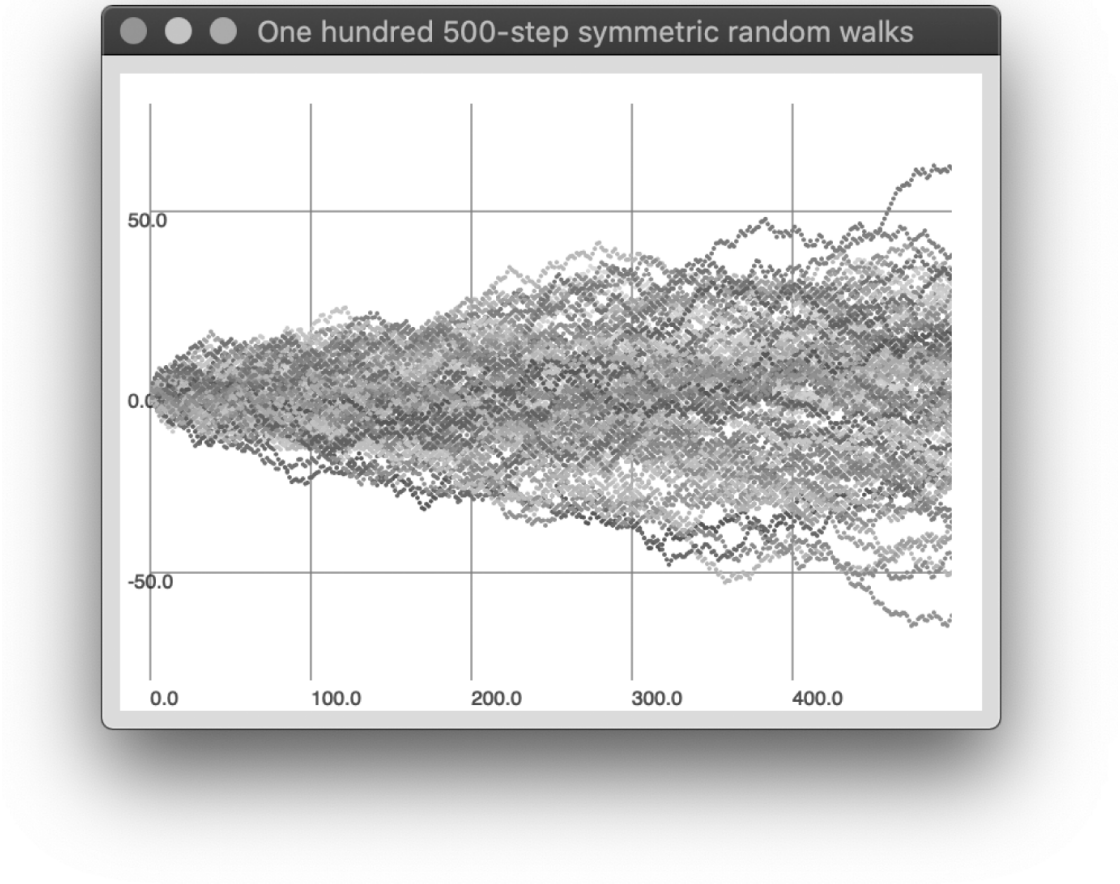


Figure 19.4
Picture of one hundred 500-step symmetric random walks.

A random walk can provide a good balance between continuity and variation. On the one hand, there is correlation between successive values, as the next value has to be within a determined distance from the current value. On the other, the position of the resulting path changes unpredictably and is unbounded.

With the method `fold`, we can add a pair of elastic barriers to a random walk to keep its values within the range that interests us for a sound parameter. The barriers will reflect excessive values back into the range. Figure 19.5 plots and plays an asymmetric random walk that generates MIDI note numbers for the `\midinote` key of a `Pbind`. The random walk is biased toward the high register (the probabilities of −1 and 1 are 45 percent and 55 percent, respectively), and it is kept within the range of the piano (from A0 to C8) with `fold`.

```
(
SynthDef(\sine, {| freq = 440, amp = 0.1, sustain = 0.25,   pan = 0
|
   var signal, env;
   signal   = SinOsc.ar(freq);
   env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
   signal = Pan2.ar(signal * env, pan);
   OffsetOut.ar(0, signal)
}).add
)
(
var walk = [60]; // middle C
500.do({
walk = walk.add((walk.last + [-1, 1].wchoose([0.45, 0.55]))   .fold
(21, 108))
});
walk.plot(discrete: true);
Pbind(\midinote, Pseq(walk), \dur, 1/50, \legato, 3, \instrument,
\sine).play
)
```

Figure 19.5
An asymmetric random walk with a pair of elastic barriers.

`Pbrown` is a pattern that implements a random walk with elastic barriers. Using the uniform distribution, it generates random steps within the range from `-step` to `step` (excluding zero). `Pbrown` returns floating-point numbers unless its `lo`, `hi`, and `step` arguments are integers. In figure 19.6, instances of `Pbrown` control the `\midinote`, `\db`, `\legato`, and `\pan` keys of two `Pbinds` that play in parallel:

```
Pbrown(lo: 0.0, hi: 1.0, step: 0.125, length: inf)
```

```
(
Ppar(
   ({
        Pbind(
        // a Pbrown with steps with a max step size of ± an // eigh
th tone (0.25)
        \midinote, Pbrown(84.0, 108.0, 0.25, inf),
        \dur, 1/25, \legato, Pbrown(0.1, 3.0, 0.3),
        \db, Pbrown(-57.0, -21.0, 6.0),
```

```
            \pan, Pbrown(-1, 1, 0.1),
            \instrument, \sine
        )
    } ! 2) // change the 2 to try different numbers of Pbinds
    ).play
    )
```

Figure 19.6
Pbrown.

In the late 1960s, Iannis Xenakis started using random walks while experimenting with new approaches to digital sound synthesis. Soon afterward, he used random walks to create melodic lines in many of his instrumental works of the 1970s and early 1980s, for example, in *Mikka* (1971) and *N'Shima* (1975) (Solomos 2001). He combined random walks with *sieves*—sequences of integer intervals that can be applied to any musical parameter—and other compositional techniques. Figure 19.7 shows three parallel random walks over the pitch sieve used by Xenakis in *Jonchaies* (1977) for full orchestra. This sieve starts at A2, has a periodicity of 17 semitones, and its intervallic structure is 1, 3, 1, 2, 4, 1, 4, and 1 semitones. Xenakis repeated the intervallic structure of the sieve until it reached E7. To achieve this, we will use the `wrapExtend` method with a `length` argument of 26 on an array containing the intervallic structure of the sieve. This will generate a new array consisting of repeated sequences of the interval structure up to size 26. Then we will use the `integrate` method to incrementally sum the intervals starting at 45 (the MIDI note number for A2) in order to obtain the corresponding MIDI note numbers.

```
(
SynthDef(\percSine, {| freq = 440, amp = 0.1, release = 0.25,   pan
= 0 |
    var signal, env;
    signal  = SinOsc.ar(freq);
    env = EnvGen.ar(Env.perc(0.01, release, amp), doneAction: 2);
    signal = Pan2.ar(signal * env, pan);
    OffsetOut.ar(0, signal)
}).add
)
(
p = Pbind(
    // a pitch sieve from A2 to E7
    \sieve, ([45] ++ [1, 3, 1, 2, 4, 1, 4, 1].wrapExtend(26))   .int
egrate,
```

```
    // a random walk that returns integers within the range of // in
dices of the pitch sieve
    \walk, Pbrown(0, 26, 1),

    // the position of the random walk (the values of \walk)
    // used as an index into the
    // array at \sieve for \midinote
    \midinote, Pfunc({|event| event[\sieve][event[\walk]]}),
    \dur, 1/8, \release, 0.5, \db, -24,
    // the values of \midinote mapped to \pan
    \pan, Pkey(\midinote).linlin(45, 100, -1, 1),
    \instrument, \percSine,
);
// three Pbinds in parallel starting after 0, 8 and 15 seconds
Ptpar([0, p, 8, p, 15, p]).play
)
```

Figure 19.7
Random walks over the pitch sieve of *Jonchaies*.

Random walks can also produce pulses with subtle variations in duration that sound organic and spontaneous:

```
Pbind(\dur, 0.25 * Pbrown(0.5, 2, 0.1125), \instrument,   \percSin
e).play
```

In figure 19.8, the melody of Steve Reich's *Piano Phase* (1967) is played in parallel by two `Pbinds` that shift out of phase with each other due to the `Pbrowns` modulating their durations.

```
(
Ppar([
    Pbind(\dur, 0.14 * Pbrown(0.975, 1.025, 0.01), \midinote,   Pseq
([76, 78, 83, 85, 78, 76, 85, 83, 78, 86, 85], inf),   \pan, -1, \i
nstrument, \percSine),
    Pbind(\dur, 0.14 * Pbrown(0.975, 1.025, 0.01), \midinote,   Pseq
([76, 78, 83, 85, 78, 76, 85, 83, 78, 86, 85], inf),   \pan, 1, \in
strument, \percSine)
]).play
)
```

Figure 19.8
The melody of Steve Reich's *Piano Phase* (1967), shifting out of phase.

We can create a sequence of random walks—an array of instances of the `Pbrown` class—and get one value at a time from each random walk cyclically with the `Pswitch1` pattern. `Pswitch1` selects elements from an array (its argument `list`) according to a stream of indices (its argument `which`). If the selected element is a stream, `Pswitch1` embeds only one value from it and selects the next element from `list` according to the next index, indicated by `which`. When a stream that has already been selected is chosen again, this stream resumes where it left off and returns its next value:

```
Pswitch1(list, which: 0)
```

Figure 19.9 shows two examples of sequences of random walks that generate pitch values. `Pswitch1` reads an array of `Pbrowns` of size `numRWalks` sequentially. The maximum size of each step is selected by weighted random choice with a `Pwrand` (in the first example, ±7 or 0 degrees with probabilities of 10 percent and 90 percent; in the second, plus or minus an eighth tone or 0 semitones with probabilities of 1 percent and 99 percent). The durations in the second example are 10 times shorter and modulated by a random walk.

```
(
// try different numbers of random walks, maximum step sizes, // pr
obabilities and durations
var numRWalks = 16;
Pbind(
   // a "just octatonic" scale: the pitch classes of the first // e
ight prime-numbered harmonics of C1 (from James Tenney)
   \scale, ((12.midicps * [2, 3, 5, 7, 11, 13, 17, 19]).   cpsmidi
 % 12).sort,
   \degree, Pswitch1(
        ({Pbrown(0, 56, Pwrand([7, 0], [0.1, 0.9], inf))} !   numRW
alks),
        Pseq(Array.series(numRWalks, 0, 1), inf)
   ),
   \octave, 2,
   \dur, 1/20, \db, −24,
   //the random walks are spread across the stereo field
   \pan, Pseq(Array.series(numRWalks, −1, 2/(numRWalks−1)), inf),
   \instrument, \percSine
).play
)
(
var numRWalks = 17;
```

```
Pbind(
   \midinote, Pswitch1(
        // floating-point numbers within the range of the piano //
 (from A0 to C8)
        ({Pbrown(21.0, 108.0, Pwrand([0.25, 0.0], [0.01,   0.99], i
nf))} ! numRWalks),
        Pseq(Array.series(numRWalks, 0, 1), inf)
   ),
   \dur, (1/200) * Pbrown(0.5, 2, 0.02), \db, −27,
   \pan, Pseq(Array.series(numRWalks, −1, 2/(numRWalks−1)), inf),
   \instrument, \percSine
).play
)
```

Figure 19.9
Sequences of random walks.

Dynamic Stochastic Synthesis (DSS) is a technique for digital sound synthesis that Xenakis designed and used in his piece *La légende d'Eer* (1977) for electronic tape. In this technique, a signal is created by repeating a waveform which is constructed by linearly interpolating a set of breakpoints. Each breakpoint is defined by a pair of amplitude and duration values. At every repetition of the waveform, each breakpoint's amplitude and duration values are varied using random walks (figure 19.10). There are as many pairs of amplitude and duration random walks as there are breakpoints in the waveform. Each random walk is forced to remain within a predefined space by means of two elastic barriers. Between the late 1980s and early 1990s, Xenakis developed a new version of this algorithm and used it in his piece *GENDY3* (1991) for electronic tape. The new version employs second-order random walks instead of first-order random walks. A second-order random walk is a random walk whose steps are generated by another random walk. For more on DSS, see Xenakis (1992, pp. 289–322), Luque (2009), and Nick Collins's Gendy UGens for SuperCollider (Collins 2011).

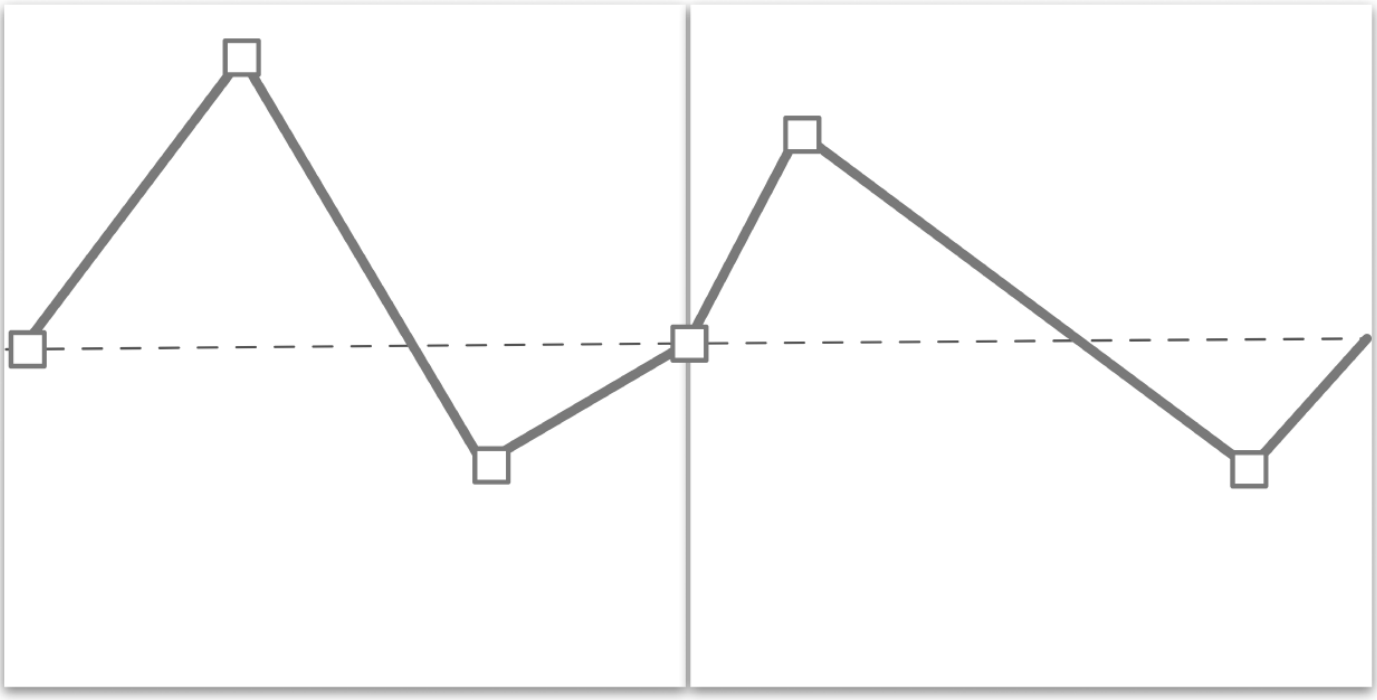


Figure 19.10
DSS: two cycles of a waveform with three breakpoints.

We will use the `DemandEnvGen` UGen to implement Xenakis's first synthesis algorithm from scratch, as this will enable us to understand it better and to vary it. For instance, after I implemented the DSS algorithm using SuperCollider, I was able to experiment with it and developed a new extension: the *Stochastic Concatenation of Dynamic Stochastic Synthesis* (Luque 2006).

`DemandEnvGen` is an envelope generator, a combination of the `EnvGen` and `Env` classes, that can obtain values for its arguments from demand-rate UGens. It generates a signal by interpolating a set of breakpoints, and each breakpoint is defined by level, duration, and shape values. (See the DemandEnvGen Help file for more details.) Demand-rate UGens allow us to have lazy evaluation inside synths, and most of these are ports of client-based patterns. `DemandEnvGen` can be very useful for nonstandard synthesis, as it allows us to create arbitrary waveforms (figure 19.11):

```
DemandEnvGen.ar(level, dur, shape: 1, curve: 0, gate: 1.0,   reset:
1.0, levelScale: 1.0, levelBias: 0.0, timeScale: 1.0,   doneAction:
0)


(
{
   // a sawtooth oscillator with DemandEnvGen
   var amps, durs;
```

```
    amps = [0, 0.025, -0.025];
    // the number of samples that each breakpoint lasts is
    // multiplied by the duration of one sample in seconds
    durs = [50, 0, 50] * SampleDur.ir;
    Pan2.ar(DemandEnvGen.ar(Dseq(amps, inf), Dseq(durs, inf)))
}.play
)
(
{
    // a waveform with 36 breakpoints
    var amps, durs, numBps;
    numBps = 36; // try different numbers of breakpoints
    // amplitudes are generated with a uniform random // distributio
n (from -0.1 to 0.1)
    amps = ({rrand(-0.1, 0.1)}! numBps);
    // durations are generated with an exponential distribution //
 (from 1 to 128 samples) and truncated to multiples of 1
    durs = ({exprand(1, 128).trunc}!numBps) * SampleDur.ir;
    Pan2.ar(DemandEnvGen.ar(Dseq(amps, inf), Dseq(durs, inf)))
}.play
)
```

Figure 19.11
DemandEnvGen.

We employ the `Dbrown` and `Dswitch1` UGens, the demand-rate versions of `Pbrown` and `Pswitch1`, to implement Xenakis's synthesis algorithm with `DemandEnvGen` (figure 19.12). Instances of `Dbrown` generate the random walks for each breakpoint's amplitude and duration values; we need two `Dbrowns` per breakpoint. For the `level` argument of `DemandEnvGen`, an instance of `Dswitch1` reads an array with the amplitude `Dbrowns` sequentially, pulling one value from each `Dbrown` at a time. For the `dur` argument, another instance of `Dswitch1` reads an array with the duration `Dbrowns` sequentially. For a detailed description of how to implement DSS algorithms, see Luque (2009).

```
(
{
    var numBps, minFreq, maxFreq, maxDur, minDur, maxDurStep,   maxA
mp, maxAmpStep;
    // number of breakpoints
    numBps = 17;
    // min and max frequencies
    minFreq = 27.5;
```

```
    maxFreq = 2500;
    // convert the min and max frequencies to number of samples
    maxDur = SampleRate.ir / minFreq;
    minDur = SampleRate.ir / maxFreq;
    // divide the min and max number of samples by the number // of
 breakpoints:
    // these values are the barriers for the duration random // walk
of each breakpoint
    maxDur = maxDur/numBps;
    minDur = minDur/numBps;
    // ± maximum size for the steps of the duration random walks
    maxDurStep = 8;
    // maximum amplitude: the ± barrier for the amplitude random //
 walks
    maxAmp = 0.1;
    // ± maximum size for the steps of the amplitude random walks
    maxAmpStep = 0.05;
    Pan2.ar(
         DemandEnvGen.ar(
               // amplitude values
               Dswitch1(
                    ({Dbrown(maxAmp.neg, maxAmp, maxAmpStep)} !   nu
mBps),
                    Dseq(Array.series(numBps, 0, 1), inf)
               ),
               // duration values
               Dswitch1(
                    ({Dbrown(minDur, maxDur, maxDurStep)} !   numBp
s),
                    Dseq(Array.series(numBps, 0, 1), inf)
               ) * SampleDur.ir
    ))
}.play
)
```

Figure 19.12
DSS employing `Dbrown` and `Dswitch1` UGens.

Figure 19.13 creates the graphical user interface (GUI) shown in figure 19.14 for controlling the parameters of the DSS algorithm.

```
(
// two DemandEnvGens (left channel and right channel)
```

```
Ndef(\Dynamic_Stochastic_Synthesis,
   {| bpRatio = 0.5, freqA = 27.5, freqB = 2093, maxDurSt =   0.1,
 maxAmp = 0.1, maxAmSt = 0.1 |
        var numBps, maxNumBps = 1603, freqs, maxDur, minDur,   maxD
urStep, maxAmpStep;
        freqs = [min(freqA, freqB), max(freqA, freqB)];
        maxDur = SampleRate.ir / freqs[0];
        minDur = SampleRate.ir / freqs[1];
        numBps = (minDur * bpRatio).trunc.clip(1, maxNumBps);
        maxDur = maxDur / numBps;
        minDur = minDur / numBps;
        maxDurStep = (maxDur - minDur) * maxDurSt;
        maxAmpStep = (maxAmp * 2) * maxAmSt;
        ({
              DemandEnvGen.ar(
                    Dswitch1(
                          {Dbrown(maxAmp.neg, maxAmp, maxAmpStep)} !
maxNumBps,
                          Dseries(0, 1) % numBps
                    ),
                    Dswitch1(
                          {Dbrown(minDur, maxDur, maxDurStep)} !   ma
xNumBps,
                          Dseries(0, 1) % numBps
                    ) * SampleDur.ir
        )} ! 2)
}).play;
// specifications for the GUI
Spec.add(\bpRatio, [0, 1, \lin, 0, 0.5]);
Spec.add(\freqA, [27.5, 2093, \exp, 0, 27.5]);
Spec.add(\freqB, [27.5, 2093, \exp, 0, 2093]);
Spec.add(\maxDurSt, [0, 1, \lin, 0, 0.1]);
Spec.add(\maxAmp, [0.001, 1, \exp, 0, 0.1]);
Spec.add(\maxAmSt, [0, 1, \lin, 0, 0.1]);
Ndef(\Dynamic_Stochastic_Synthesis).gui
)
```

Figure 19.13
Code for a GUI for DSS.

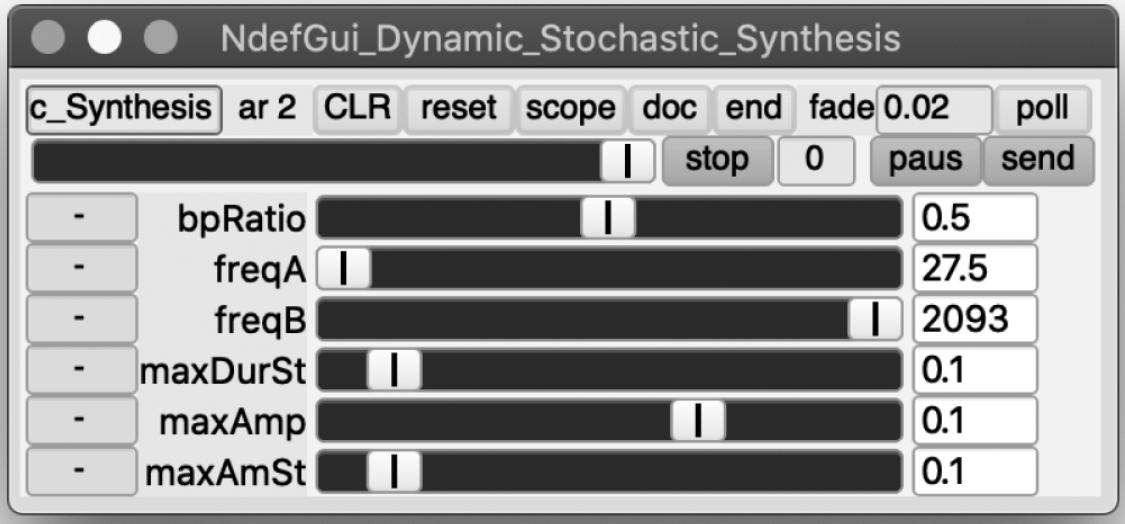


Figure 19.14
Picture of a GUI for DSS.

19.2 Tendency Masks

In a tendency mask, random values are chosen between boundaries that change over time. This technique was one of the *selection principles* used by Gottfried Michael Koenig in his programs Project 2 (1966–1968), for instrumental composition, and *SSP,* designed by Koenig in 1972 and implemented by Paul Berg in 1977, for sound composition (Koenig 1970; Berg 1979). Several composers have utilized Koenig's technique, for instance, Barry Truax in his POD programs for digital sound synthesis (Truax 1973).

Tendency masks can create shapes that are perceivable in different parameters. This allows us to orient ourselves: we can know where we have come from and where we are probably going. As listening to tendency masks involves memory and prediction, they are valuable because they provide a way to generate direction and tension that makes use of random numbers.

To implement a tendency mask, we need to specify the parameters of the pair of breakpoint envelopes that will define the lower and upper boundaries for the random selections. We will generate these envelopes in the client with the `Pseg` pattern, and in the server with the `DemandEnvGen` UGen.

`Pseg` samples a breakpoint envelope as a function of time, and it is useful for controlling parameters that change over time from the client. It is based on `Env.new` and can obtain the values for its first three arguments from patterns (at least, the first argument `levels` should be a pattern):

```
Pseg(levels, durs, curves, repeats)
```

In figure 19.15, a `Pseg` goes from C4 to G4 in 5 s and back to C4 in 1 s.

```
(
Pbind(
   // from C4 to G4 in 5 seconds and back to C4 in 1 second
   \midinote, Pseg(Pseq([60, 67] ,inf), Pseq([5, 1] ,inf)).   trac
e,
   \dur, Pexprand(0.1, 0.5),
   \instrument, \percSine
).play
)
```

Figure 19.15
`Pseg`.

Figure 19.16 gives examples of tendency masks for pitch, panning position, and duration in `Pbinds`. `Pwhite` generates random values from the uniform distribution between the boundaries created by a pair of `Psegs`. Figure 19.17 shows the tendency mask for pitch values of the first example in figure 19.16.

```
(
SynthDef(\saw, {| freq = 440, amp = 0.1, sustain = 0.25,   pan = 0
 |
   var signal, env;
   signal = Saw.ar(freq);
   env = EnvGen.ar(Env.sine(sustain, amp), doneAction:2);
   signal = Pan2.ar(signal * env, pan);
   OffsetOut.ar(0, signal)
}).add
)
(
// tendency masks for pitch and panning (2100 notes)
~allMidinotes = [];
Pbind(
   \midinote, Pwhite(
```

```
        // from B5 to C8 to C#8 to C#1 in 12, 2 and 3 seconds
        Pseg(Pseq([83, 108, 109, 25, 25]), Pseq([12, 2,   3, 4])),
        // from F#6 to C8 to C#8 to F#2 in 12, 2 and 7 seconds
        Pseg(Pseq([90, 108, 109, 42]), Pseq([12, 2, 7]))
    ),
    \pan, Pwhite(
        // from left to right to left in 14 and 3 seconds
        Pseg(Pseq([−1, 1, −1, −1]), Pseq([14, 3, 4])),
        // from center to right in 14 seconds
        Pseg(Pseq([0, 1, 1]), Pseq([14, 7]))
    ),
    \dur, 1/100, \legato, 25, \db, −27,
    \instrument, \saw,
    \addToList, Pfunc({| event | ~allMidinotes = ~allMidinotes.   ad
d(event[\midinote])})
).play
)
// plot the values generated by the tendency mask for the // \midin
ote key
~allMidinotes.plot(discrete: true).resolution_(0)
(
// a tendency mask for pitch
// based on James Tenney's "Form 4 - in memoriam Morton
// Feldman" for chamber ensemble (five times faster: this
// example lasts 3'15")
~allMidinotes = [];
Pbind(
    // seven notes from the octatonic scale (pitch class set 7-31)
    \scale, [0, 1, 3, 4, 6, 7, 10],
    \degree, Pwhite(
        Pseg(Pseq([1, 33, 33]), Pseq([810, 165]/5)),
        Pseg(Pseq([0, 0, 32, 32]), Pseq([330, 615, 30]/5))
    ).round,
    \root, 4, \octave, 3,
    \dur, Pwhite(0.1875, 0.375), \legato, 16, \db, −36,
    \pan, Pseq([−1, −0.33, 0.33, 1], inf),
    \instrument, \sine,
    \addToArray, Pfunc({| event | ~allMidinotes =~allMidinotes.add
(event.use({~midinote.value}))})
).play
)
// plot the values generated by the tendency mask as MIDI note // n
umbers
~allMidinotes.plot(discrete: true).resolution_(0)
```

```
(
// a tendency mask for duration
// loosely based on the final part of Elliott Carter's 90+ for // s
olo piano
Pbind(
   // durations go from disorder to order and back to disorder
   \dur, Pwhite(
        Pseg(Pseq([0.016, 0.125, 0.125, 0.016]), Pseq([15,   3, 1
5])),
        Pseg(Pseq([3, 0.125, 0.125, 3]), Pseq([15, 3, 15]))
   ),
   // hexachord 6-Z17 (inverted) is permutated randomly and // tran
sposed cyclically (+3 and +8 semitones) at each // repetition
   \note, Pn(Pshuf([0, 1, 4, 6, 7, 8], 1), inf) +   Pseq([Pn(3, 6),
Pn(8, 6)], inf),
   \instrument, \percSine
).play
)
```

Figure 19.16
Tendency masks for pitch, panning, and duration.

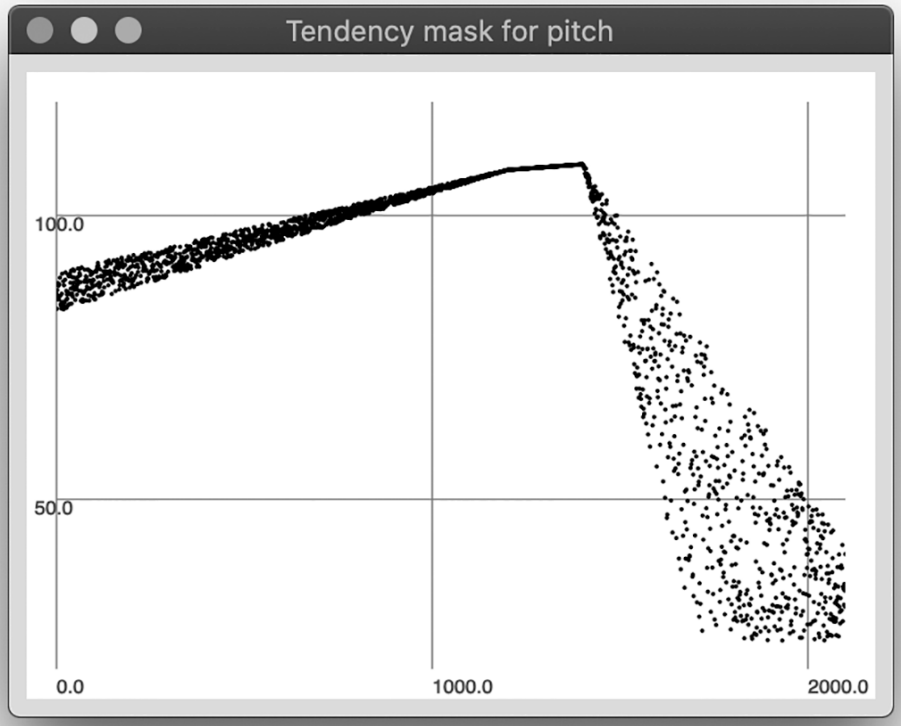


Figure 19.17
Tendency mask for pitch values of the first example from figure 19.16.

Tendency masks are useful to articulate sound masses, to create shapes that change over time in the pitch register, to establish different speeds and different degrees of order and disorder in duration sequences, to produce values quickly that convey a sense of direction for granular synthesis, and to generate transitions and different levels of stability in nonstandard synthesis. At the same time, we learned the hard way that simple shapes tend to give the best results.

Tendency masks are one of the random procedures used in SSP to join amplitude and duration values into waveforms (*segments*) and to determine the order of the waveforms (Berg 1979). Berg (2009, p. 84) remarks about it: "The ordering of segments using tendency masks was particularly successful. A wide selection of segments would result in a noisy sound structure. Narrow masks led to unstable sounds within a confined frequency region. Masks moving from narrow to wide could produce dramatic transitions between these two extremes."

With `DemandEnvGen`, we can create waveforms with random amplitude and duration values and concatenate them in an order determined by a tendency mask implemented

with demand-rate UGens: `Dwhite` here generates random values between the boundaries created by two `DemandEnvGens` (figure 19.18).

```
(
// a function that creates arrays with random amplitude and // dura
tion values
// DemandEnvGen will create waveforms with these values
~waveformMaker = {arg numWaveforms = 12, loPitch = 21.0,   hiPitch
= 96.0;
   ~waveforms= ({
        var numBps, amps, totalDur, durs;
        totalDur = rrand(loPitch, hiPitch).min(96).midicps.   recip
rocal * s.sampleRate;
        numBps = (totalDur * exprand(0.1, 0.25)).asInteger;
        amps = [0] ++ ({rrand(-1.0, 1.0)} ! (numBps - 1));
        durs = totalDur.round.partition(numBps);
        [amps, durs, numBps]
   } ! numWaveforms)
}
)
({
   // one waveform
   var amps, durs;
   ~waveformMaker.value(1); // the function creates the   // parame
ters of one waveform
   #amps, durs = ~waveforms.flop;
   Pan2.ar(DemandEnvGen.ar(Dseq(amps[0], inf), Dseq(durs[0]   * Sam
pleDur.ir, inf)), 0, 0.1)
}.play
)
({
   // 4 waveforms concatenated randomly with a Drand
   var amps, durs, numBps, index, selection;
   ~waveformMaker.value(4); // change the number of waveforms
   #amps, durs, numBps = ~waveforms.flop;
   amps = amps.collect({| waveformAmps | Dseq(waveformAmps,   in
f)});
   durs = durs.collect({| waveformDurs | Dseq(waveformDurs,   in
f)});
   selection = Ddup(2, Drand(Array.series(~waveforms.size, 0, 1),
inf));
   index = Ddup(Dswitch1(numBps, selection) * 2, selection);
```

```
   Pan2.ar(DemandEnvGen.ar(Dswitch1(amps, index), Dswitch1(durs   *
SampleDur.ir, index)), 0, 0.25)
}.play
)
({
   // a narrow tendency mask: 2 waveforms
   var amps, durs, numBps, index, tendency;
   ~waveformMaker.value(2);
   #amps, durs, numBps = ~waveforms.flop;
   amps = amps.collect({| waveformAmps | Dseq(waveformAmps,   in
f)});
   durs = durs.collect({| waveformDurs | Dseq(waveformDurs,   in
f)});
   tendency = Ddup(2,
        Dwhite(
              DemandEnvGen.ar(Dwhite(0.5, 1.0, inf), 4),
              DemandEnvGen.ar(Dwhite(1.0, 1.5, inf), 4),
        ).trunc
   );
   index = Ddup(Dswitch1(numBps, tendency) * 2, tendency);
   Pan2.ar(DemandEnvGen.ar(Dswitch1(amps, index), Dswitch1(durs   *
SampleDur.ir, index)), 0, 0.25)
}.play
)
({
   // a tendency mask that goes from narrow to wide and back // to
 narrow
   var amps, durs, numBps, index, tendency;
   ~waveformMaker.value(6, 60.0, 96.0);
   #amps, durs, numBps = ~waveforms.flop;
   amps = amps.collect({| waveformAmps | Dseq(waveformAmps,   in
f)});
   durs = durs.collect({| waveformDurs | Dseq(waveformDurs,   in
f)});
   tendency = Ddup(2,
        Dwhite(
              DemandEnvGen.ar(
                   Dseq([0, 0, 0, ~waveforms.lastIndex + 0.99]),
                   Dseq([2, 3, 3])
              ),
              DemandEnvGen.ar(
                   Dseq([0, 0, ~waveforms.lastIndex + 0.99,   ~wave
forms.lastIndex + 0.99]),
```

```
                    Dseq([2, 3, 3])
                )
            ).trunc
    );
    index = Ddup(Dswitch1(numBps, tendency) * 2, tendency);
    Pan2.ar(DemandEnvGen.ar(Dswitch1(amps, index), Dswitch1(durs   *
SampleDur.ir, index)), 0, 0.25)
}.play
)
({
    // a tendency mask with breakpoints with random levels
    var size, amps, durs, numBps, index, tendency;
    size = rrand(3, 12);
    postf("% waveforms\n", size);
    ~waveformMaker.value(size);
    #amps, durs, numBps = ~waveforms.flop;
    amps = amps.collect({| waveformAmps | Dseq(waveformAmps,   in
f)});
    durs = durs.collect({| waveformDurs | Dseq(waveformDurs,   in
f)});
    tendency = Ddup(2,
        Dwhite(
            DemandEnvGen.ar(Dwhite(0, ~waveforms.lastIndex +   0.
99), 4),
            DemandEnvGen.ar(Dwhite(0, ~waveforms.lastIndex +   0.
99), 4)
        ).trunc
    );
    index = Ddup(Dswitch1(numBps, tendency) * 2, tendency);
    Pan2.ar(DemandEnvGen.ar(Dswitch1(amps, index), Dswitch1(durs   *
SampleDur.ir, index)), 0, 0.25)
}.play
)
```

Figure 19.18
Tendency masks for the concatenation of waveforms.

The following sections of this chapter present three synthesis strategies and their implementations in SuperCollider:

1. Synthesis with integer operations
2. Buffer modulation (buffer scratching)
3. Buffer rewriting

These strategies have some characteristics in common: there exists little academic theory on them, and they are very easy to implement in plain SuperCollider without any extensions. As a welcome consequence, they immediately offer plenty of opportunities for sound experimentation and exploration in unchartered territory. Due to the universality of the approaches, sounding results can be within a wide range. That is especially true for (2) and (3) as buffers can be filled and rewritten with any audio data, whether synthesized or recorded.

19.3 Synthesis with Integer Operations

19.3.1 The Bytebeat Movement

In 2011 the so-called Bytebeat movement occurred as an online phenomenon, triggered by the Finnish computer artist Ville-Matias Heikkilä (aka Viznut). A set of seven short sound-producing C-language programs was presented in a YouTube video.[1] In turn, several individuals contributed similar examples.[2,3] Heikkilä (2011) describes this uncoordinated explorative process,[4] where he also delivers analytical work. Jonathan Siemasko (aka schemawound)[5] also elaborates on bitwise operation in his SuperCollider code archive. Jeffrey Morris puts the method into a historical context.[6]

In its original form, an integer counter *t*, proceeding at a rate of 8,000 Hz, is used for combinations of bitshifts and bitwise logical and basic arithmetic operations (*not, and, or, +, *, div, mod,* etc.). The result is mapped to 8 bits (a grid of 256 values) and normalized.

As Heikkilä shows,[7] self-similar structures like the Sierpinski harmony emerge from the astonishingly simple one-liner `t & (t >> 8)`. The symbol `&` stands for the bitwise and operation, and >> stands for the bitshift to the right; the compound function then applies to the integer counter `t` at each sample (respectively, at each new counter if optimized). The self-similarity in this example unfolds within some seconds. The process might take a long time to loop in more complicated cases. That offers the promising possibility of harvesting musical forms as emerging results of the deterministic synthesis procedure.

The original Bytebeat examples intentionally commit to the 8-bit *chiptune* aesthetics of early computer games. However, with the application of SC's capabilities of modulation and transformation, the sounding results can go in different directions. In any case, by tweaking prime number relations, one can keep the tendency toward slowly unfolding developments.

To implement the original examples in SC, we need to consider the following three requirements:

1. A counter at 8,000 Hz and a sample rate of 48 kHz

2. A reduction of amplitude to 8-bit resolution (using modulo 256) and normalization
3. Adding parentheses, as the order of execution is different in SC

A sample rate of 48 kHz is not a must, but a recommendation to get very close to the timbre of the original bytebeat examples with the SC emulations below. As 8,000 is not a divisor of 44,100, irregularities can result in additional audible distortion with a sample rate of 44.1 kHz. However, readers might want to experiment with 44.1 kHz and a counter at dividing rates (e.g., 7,350 or 8,820 Hz). Similarly, alternative or varying bit resolutions are an option.

Figure 19.19 shows a straightforward implementation in SC. The bit operations are applied at each sample. In an optimization attempt, one could implement sample-and-hold by using small block sizes (4 or 8) and running a control rate ramp (e.g., Sweep). As the calculations are not costly anyway—and for simplicity—this option is omitted in the following examples.

```
// first example from https://www.youtube.com/watch?v=tCRPUv8V22o
// the posted input doesn't work in SC because of LR-precedence!
// (t * 5 & t >> 7) | (t * 3 & t >> 10)

(
x = {
   var t = PulseCount.ar(Impulse.ar(8000)); // assuming server   ru
nning 48 kHz!
   var sig = ((t*5) & (t>>7)) | ((t*3) & (t>>10));
   (sig % 256).linlin(0, 255, 0, 0.2)! 2 // reduce to 8 bit   and l
imit
}.play
)
x.release
```

Figure 19.19
Example by Viznut.

To abstract the building process, we can write a maker `Function` that produces a `UGen` graph `Function` (figure 19.20). While playing audio examples, one might want to observe the fascinating self-similar structures in the scope window in their original form. In general, however, the application of `LeakDC` is recommended to gain headroom for the signal. For that reason, the maker `Function` includes leakage and lowpass options.

```
(
~bytebeats = {|func, rate = 8000, mod = 256, amp = 0.3, cutoff,   l
eakDC = false|
   {
   |att = 0.1, rel = 1, gate = 1|
   var t = PulseCount.ar(Impulse.ar(rate.value));
   var sig = func.(t) % mod.value;
   var env = EnvGen.ar(Env.asr(att, 1, rel), gate, doneAction: 2);

   // NamedControl allows using default arguments passed to the //
 maker function
   var amplitude = \amp.kr(amp.value);
   var maxAmp = 0.5;

   // map to [0, amp], amp is kept below maxAmp
   sig = sig.linlin(0, mod, 0, amplitude.clip(0, maxAmp));

   // LPF, LeakDC
   sig = cutoff.notNil.if {LPF.ar(sig, cutoff.value)} {sig};
   sig = leakDC.if {LeakDC.ar(sig)} {sig};

   // allow multichannel handling, expand mono to stereo
   (sig.size > 1).if {sig} {sig! 2} * env
   }

};

s.scope;
)
// define bytebeat as Function and use it as input to the // maker
 Function
f = {|t| ((t*5) & (t>>7)) | ((t*3) & (t>>10))};

// assume 48 kHz
u = ~bytebeats.(f, 8000);

x = u.play;
x.set(\amp, 0.1);
x.release;

// use the UGen graph Function with Ndef
```

```
Ndef(\b, u).play;
Ndef(\b).release;
```

Figure 19.20
Code that makes a `UGen` graph `Function`.

19.3.2 Extending and Generalizing Bytebeats

We can apply modulation to the fixed parameters of the original bytebeat concept. With the above `Function` maker, one can pass LFO (low frequency oscillator) aggregates (figure 19.21).

```
f = {|t| t * (42 & (t>>10))};
x = ~bytebeats.(f, amp: 0.3).play;
x.release

// stereo variant with oscillating rate
// the maker function args can be passed ugen aggregates, // wrappe
d in a function

(
x = ~bytebeats.(f,
   rate: {SinOsc.ar(SinOsc.ar(0.2).range(5, 20), [0, pi]).   range
(7500, 8500)}
).play
)

x.release

// modulated bit(crushing) param, stereo
(
x = ~bytebeats.(f,
   mod: {2 ** (SinOsc.ar(LFSaw.ar([0.2, 0.21]).range(0.2, 200)).
 range(4, 8).round)}
).play
)
x.release
```

Figure 19.21
Variants of the "42 Melody," cited by Viznut.

Instead of an integer counter, one can choose alternative sources. Linear `UGens` like `Sweep`, `Phasor`, `Saw`, `VarSaw`, and `LFDNoise1` are closest to the idea of bytebeats, but other sources can be tried as well (figure 19.22). With deterministic `UGens`, slight

deviations from integer relations are often a promising strategy to achieve interesting macro developments (second example in figure 19.22).

```
// linear UGens can chime in instead of a counter:
// Sweep, Phasor, Saw, VarSaw, LFDNoise1 etc.
(
x = {
   var sig = LFDNoise1.ar(5! 2).range(1000, 15000).round;
   sig = sig & (sig << 7);
   LeakDC.ar(sig.linlin(0, 1024, 0, 0.2)) // linear mapping   clips
values over 1024
}.play
)
x.release

// varying the width of a VarSaw
(
x = {
   var sig = VarSaw.ar(2, width: SinOsc.ar([5.49, 5.51],   [0, p
i]).range(0, 1)).range(500, 1500).round;
   sig = sig & (sig << 7);
   LeakDC.ar(sig.linlin(0, 1024, 0, 0.2)) // linear mapping   clips
values over 1024
}.play
)
x.release
```

Figure 19.22
Alternatives to an integer counter.

The greatest common divisor `gcd` and the least common multiple `lcm` are further rewarding options. They can be part of more classical bytebeat setups or act generically, e.g., after a modulo operation (figure 19.23). Many other examples with these operators emerged from a thread initiated by Nick Collins on the old SuperCollider mailing list.[8]

```
// These operations are related:
// gcd(a, b) = a * b / lcm(a, b)
// NOTE: 'gcd'/'lcm' apply 'floor',
// so there might occur differences when applying 'round' to a // c
ontinuous signal!

// gcd used for synthesis on different counters
```

```
// it can perform multichannel expansion which opens nice // opport
unities
(
x = {
   var t = PulseCount.ar(Impulse.ar(3000)) % 100;
   var u = PulseCount.ar(Impulse.ar(2995)) % 100;
   var sig = gcd(t, u + [30, 25]);
   Limiter.ar(LPF.ar(HPF.ar(sig, 20), 10000), 0.3)
}.play
)
x.release

// triggering can be done with demand rate ugens to get finer // co
ntrol
// iterated gcd on 3 counters
(
x = {
   var a = TDuty.ar(Dseq((1..10), inf) + 102 * SampleDur.ir);
   var b = TDuty.ar(Dseq((1..10), inf) + 101 * SampleDur.ir);
   var c = TDuty.ar(Dseq((1..10), inf) + 100 * SampleDur.ir);
   var t = PulseCount.ar(a) % 100;
   var u = PulseCount.ar(b) % 101;
   var v = PulseCount.ar(c) % 102;
   var sig = gcd(gcd(t + [0, 0], u + [0, 10]), v + [0, 20]);

   // or, better, especially with more iterations:
   // var sig = [t + [0, 0], u + [0, 10], v + [0, 20]]. // reduce
(\gcd);
   Limiter.ar(LPF.ar(HPF.ar(sig, 20), 10000), 0.3)
}.play
)
x.release
```

Figure 19.23
Examples with the greatest common divisor and least common multiple.

Another related process is *bit scrambling* (or *bit mangling*—credit to Benedikt Alphart for the hint), where the bits are reordered according to some permutation array. A scrambling implementation in SC requires more care than the basic bit operations. Ultimately, we want to control the scrambling data, which suggests the usage of an array argument (figure 19.24).

```
~bus = Bus.audio(s, 2);
(
// start fx before source
// use LPF as scrambling can cause much energy in higher parts // o
f spectrum

y = {|amp = 0.05, cutoff = 10000|
   // more comfortable to write array arg with NamedControl
   var scramble = NamedControl.ar(\scramble, (0..7));
   var sig = In.ar(~bus, 2);
   var bits, mod_bits, sound;
   sig = sig.round;
   // we want the bits in an array
   mod_bits = 8.collect {|i|
        // scramble[i] indicates the bit value at *mapped* // posit
ion
        // >> here means division by power of 2
        sig & (2 ** scramble[i]) >> scramble[i]
   };
   // "re-synthesis" from bit representation
   sound = mod_bits.sum {|val, i| 2 ** i * val} / 256;
   Limiter.ar(LPF.ar(LeakDC.ar(sound), cutoff), amp)
}.play
)
// start source
x = {Out.ar(~bus, SinOsc.ar([50, 50.1], 0, 0.2, 1) * 128)} .play;

// define permutation of bits
y.set(\scramble, (0..7).swap(0, 3));

// evaluate several times
y.set(\scramble, (0..7).scramble.postln);

// non-permutation is also ok
y.set(\scramble, [1, 0, 3, 3, 4, 6, 1, 4]);

y.release;
x.free;
```

Figure 19.24
Basic bit scrambling by passing permutation arrays.

Multi-LFO control of bit arrays is also an attractive option. Figure 19.25 shows a continuous variation of index mapping without strict permutations.

```
// index mapping can also be controlled by arbitrary LFOs
(
x = {
   // source with decorrelated amp LFOs in range [0, 256]
   var sig = LFSaw.ar([60, 60.1], 0, LFDNoise3.ar(0.1! 2).   range
(0.2, 0.7), 1) * 128;
   var bits, mod_bits, sound;
   sig = sig.round;
   mod_bits = 8.collect {|i|
        // bit mapping LFOs with different speed
        var mod_bitIndex = SinOsc.ar(0.2 * (i + 1)).range(0, 7);
        sig & (2 ** mod_bitIndex) >> mod_bitIndex
   };
   sound = mod_bits.sum {|val, i| 2 ** i * val} / 256;
   Limiter.ar(LeakDC.ar(sound), 0.1)
}.play
)
x.release
```

Figure 19.25
Dynamic bit mapping

19.4 Buffer Modulation (Buffer Scratching)

In classical waveshaping, a mathematical function—called the *transfer function*—is used to map values of an input signal, in the normalized case from the interval [−1, 1] onto itself (Roads 1996, pp. 252–261). Nonlinear functions can lead to highly distorted signals, even with sine sources, and aliasing is likely to appear. Aside from some well-examined special functions like Chebychev polynomials, waveshaping is hard to describe mathematically, which forces the user to engage in individual experimentation.

Technically, buffer modulation is nothing more than waveshaping with a given buffer used as a transfer function. Any source can act as buffer content, but we use a recording in the examples below to help distinguish some possibilities. With a low-frequency, zigzag input signal, such buffer modulation is a digital analog to a scratching turntablist, hence the term *buffer scratching*. Buffer modulation can also be interpreted as generalized table-lookup synthesis with a nonlooping indexing signal (Puckette 2007, pp. 29–30). Regarded from still a different angle, it is a form of phase modulation that, if fast enough, can produce typical sideband effects.

We consider `BufRd` as the most flexible UGen for implementation in SC. `Shaper` would also be possible, but it requires dealing with the `Wavetable` format and has no significant performance advantage (equivalently, `PlayBuf` is also an option). Its rate argument can be obtained from the `BufRd` phase input by derivation, and vice versa by integration (figure 19.26). However, with `BufRd`, the signal to be shaped (the buffer modulator) can be passed directly; it only requires scaling to the length of the buffer. A further advantage of `BufRd` is its interpolation type option.

```
// SC standard soundfile #1
(
~path = Platform.resourceDir +/+ "sounds/a11wlk01-44_1.aiff";
~buf = Buffer.read(s, ~path);
)

// scratching between buffer indices 2000 and 4000
{BufRd.ar(1, ~buf, SinOsc.ar(3, 0, 1000, 3000))} .plot(0.1);
// equivalent: PlayBuf, needs slope/rate (deviation of index // cha
nge)
// rate in seconds, thus divide by sample rate
{PlayBuf.ar(1, ~buf, Slope.ar(SinOsc.ar(3, 0, 1000)) /   SampleRat
e.ir, 1, 3000, 1)} .plot(0.1)

// vice versa: rate oscillation given
{PlayBuf.ar(1, ~buf, SinOsc.ar(10, 0, 0.2), 1, 1000, 1)} .   plot
(0.1)

// equivalent: Integrator sums up all samples, no need to // regard
sample rate
{BufRd.ar(1, ~buf, Integrator.ar(SinOsc.ar(10, 0, 0.2)) +   1000)}
 .plot(0.1);
```

Figure 19.26
Equivalence of buffer modulation with `BufRd` and `PlayBuf`.

The input of buffer modulation can be an addition of a (local) zigzag—fast enough to generate an audible result of the shaping procedure—and a slower (global) movement through the buffer (figure 19.27). This model relates to a common technique of buffer granulation, where the center position of a granular cloud can move similarly.

```
// decorrelated local movement
(
x = {
```

```
   var sig = BufRd.ar(1, ~buf,
        phase: (
              LFDNoise3.ar(0.3).range(0.1, 0.9) + // global   movem
ent
              SinOsc.ar(100).range(0, [0.003, 0.0032]) // local   m
ovement
        ) * BufFrames.ir(~buf),
   interpolation: 4
) * 0.2;
 LeakDC.ar(sig)
}.play
)
x.release
```

Figure 19.27
Global and local movement.

Either of the movements can be a decorrelated stereo signal. As `BufRd` performs multichannel expansion, the implementation is straightforward (figure 19.28).

```
// use Sweep (linear raise) and Fold to generate zig-zag
(
x = {
   var sig = BufRd.ar(1, ~buf,
        phase: (
              // global movement
              LFDNoise3.ar(0.3) * [1, 1.01] +
              // local movement
              Fold.ar(Sweep.ar(0, LFDNoise3.ar(1).range(0.5, 1)),
hi: 0.12)
        ) * BufFrames.ir(~buf),
        interpolation: 4
) * 0.2;
LeakDC.ar(sig)
}.play
)
x.release

// more extreme zig-zag changes by LFDNoise0 (step noise) for // ar
bitrary rates
// frequency of step noise is also changing
(
x = {
```

```
    var sig = BufRd.ar(1, ~buf,
         phase: (
               // global movement
               LFDNoise3.ar(0.3) * [1, 1.01] +
               // local movement
               Fold.ar(
                    Sweep.ar(
                         0,
                         LFDNoise0.ar(
                               LFDNoise3.ar(0.3).exprange(0.3, 35)
                         ).exprange(0.1, 5)
                    ),
                    hi: 0.02
               )
    ) * BufFrames.ir(~buf),
         interpolation: 4
) * 0.2;
    LPF.ar(LeakDC.ar(sig), 10000)
}.play
)
x.release
```

Figure 19.28
Variants of global and local movement.

As an alternative model, the buffer modulation signal can be the sum of a principal modulator—already able to produce audible frequencies—and a disturbance. If the latter contains discontinuities, this will result in perceivable clicks. The example in figure 19.29 varies the smoothing of the discontinuous `LFPulse`, which results in a controlled warp effect.

```
(
x = {
    var mainPhase = LFTri.ar(
         LFDNoise0.ar(LFDNoise1.ar(1).range(0.2, 1)).range(0.2, 0.3)
    ).range(0, BufFrames.ir(~buf));

    var disturbance = LFPulse.ar(
         LFDNoise1.ar([2, 2.01]).range(3, 12),
         mul: 10000
    ).lag(LFDNoise3.ar(1).exprange(0.0001, 0.05));
    var sig = BufRd.ar(1, ~buf, phase: mainPhase + disturbance,   in
terpolation: 4) * 0.5;
```

```
    LPF.ar(LeakDC.ar(sig), 12000)
}.play
)
x.release
```

Figure 19.29
Model of main modulation and disturbance.

Keeping in mind that any signal can serve as a `BufRd` phase input, the example shown in figure 19.30 uses `DemandEnvGen`, which—due to its sequencing capabilities—is a promising candidate for modulation.

```
// with the first parameters for dur and range, this example // is
 close to vinyl scratching
// the commented out preset results in totally different // charact
eristics
(
x = {
    var dur = 0.2, range = 10000;
    // var dur = 0.02, range = 1000;

    // LR decorrelation
    var phases = {
         DemandEnvGen.ar(
               Dseq([1, 2, 1, Drand([2, 3], 1)] * range, inf),
               dur,
               shape: Drand([1, 2, 3, 4], inf) // sequence of   curv
atures
         )
    }! 2;
    BufRd.ar(1, ~buf, phase: phases, interpolation: 4) * 0.5
}.play
)
x.release
```

Figure 19.30
Modulation with `DemandEnvGen`.

19.5 Buffer Rewriting

We now outline a method for the curious who want to surprise themselves! The idea of the procedure is simple: one can write to and read from a buffer simultaneously at different or changing rates. As a result, the input signal gets scrambled in unpredictable

ways. The audible effect appears like a messed-up delay line. One place where the technique has previously been mentioned is in a workshop by the British electronic musician Chris Jeffs (aka Cylob) in Barcelona in 2012.[9]

First, let's regard under which conditions rewriting pipes audio data through the buffer without any alteration (but, in general, with delay). If writing comes before reading in the `SynthDef` graph and the read and write pointers are just looping through the buffer with a rate of 1, then the written and the read signal are identical. Things become interesting if at least one of these conditions is not guaranteed.

As a simple example, regard a buffer with a size of 8. Let's further assume a block size of 1; as writing and reading happen in blocks, a phase index distance below the designated block size inhibits the reading of previous data, even for unequal indices. If the reading rate equals 3, then the first samples of the output signal stem from the buffer indices 0 3 6 1 4 7 2 5. If the writing rate equals 1, the output will consist of scrambled input samples. On the other hand, if the writing rate equals 3, the output signal will remain unchanged. Note that `BufWr`—in contrast to `BufRd`—doesn't interpolate. Consequently, the process can lead to an alteration of the input signal, even with identical noninteger rates.

Some further considerations: if the writing or reading rate is a divisor of the buffer length and > 1, not all buffer positions are involved in writing or reading. For noninteger writing rates, the overwriting of the whole buffer might be incomplete, or it happens only after quite a long time. Hence, the result can gain the characteristic of an irregularly blurred delay line.

Regarding reading and writing together, it becomes clear that particularly unequal noninteger rates of pointers for reading and writing can lead to totally opaque scrambling and delay effects. That applies even more if rates are dynamically changing or arbitrary (nonphasor) read/write signals are chosen. The latter option also connects the principles of buffer modulation and rewriting. Figure 19.31 shows basic examples of changing the rates for writing and reading with a sine source.

```
// distorting a sine wave
(
~buf = Buffer.alloc(s, 1000, 1);
~bus = Bus.audio(s, 1);
s.scope;
s.freqscope;
)

// start fx
(
x = {|readRate = 1, writeRate = 1|
```

```
   var inSig = In.ar(~bus, 1);
   var writeOffset = 500; // offset above blockSize
   var read, write;

   write = BufWr.ar(inSig, ~buf, Phasor.ar(0, writeRate, 0,   BufFr
ames.ir(~buf)) + writeOffset);
   read = BufRd.ar(1, ~buf, Phasor.ar(0, readRate, 0,   BufFrames.i
r(~buf)));
   Limiter.ar(read, 0.2)! 2
}.play
)

// start source, no change as rates equal 1
y = {Out.ar(~bus, SinOsc.ar(200, 0, 0.1))} .play

// distortion with different rates
x.set(\readRate, 1.2)
x.set(\writeRate, 0.93)
x.set(\readRate, 0.75)
x.set(\writeRate, 2.17)
x.set(\readRate, 0.49)

s.freeAll
// control of write and read rates with MouseX and MouseY
// clear buffer from old data first
~buf.zero

// start fx
(
x = {
   var inSig = In.ar(~bus, 1);
   var read, write;

   write = BufWr.ar(inSig, ~buf, Phasor.ar(0, MouseX.kr(0.2, 2),
 0, BufFrames.ir(~buf)));
   // stereo by slightly different reading rates
   read = BufRd.ar(1, ~buf, Phasor.ar(0, MouseY.kr(0.2, 2) *   [1,
 1.01], 0, BufFrames.ir(~buf)));
   Limiter.ar(read, 0.2)
}.play
)

// start source
```

```
y = {Out.ar(~bus, SinOsc.ar(50, 0, 0.1))} .play
y.free

// source with moving frequency
y = {Out.ar(~bus, SinOsc.ar(LFDNoise1.ar(3).exprange(50, 100),   0,
0.1))} .play
// cleanup
s.freeAll

(
~buf.free;
~bus.free;
)
```

Figure 19.31
Basic changes of writing and reading rates.

The size of the buffer also plays an important role: on average, the larger it is, the longer it takes to overwrite all the values. One can limit the number of samples used within a given buffer by setting the bounds of the phase inputs (figure 19.32).

```
(
~buf = Buffer.alloc(s, 5000);
~bus = Bus.audio(s, 1);
)

// the length argument determines the relative length of used // bu
ffer section
(
SynthDef(\rewrite_var_length, {|out, readRate = 1, writeRate = 1,
   length = 1, cutoff = 5000, amp = 0.5|
   var inSig = In.ar(~bus, 1);
   var read, write;
   write = BufWr.ar(inSig, ~buf, Phasor.ar(0, writeRate, 0,   BufFr
ames.ir(~buf) * length));
   read = BufRd.ar(1, ~buf, Phasor.ar(0, readRate * [1, 1.01], 0,
BufFrames.ir(~buf) * length));
   Out.ar(out, Limiter.ar(LPF.ar(read, cutoff), amp))
}, metadata: (
   specs: (
        readRate: [0.2, 2, \lin, 0, 1],
        writeRate: [0.2, 2, \lin, 0, 1],
        length: [0.1, 1, 5, 0, 1],
```

```
            cutoff: [50, 16000, \exp, 0, 5000],
            amp: [0, 0.5, \db, 0, 0.5]
        )
    )
    ).add
    )

    // start fx with GUI before source
    // distortion with read/write rate values unequal to 1
    SynthDescLib.global[\rewrite_var_length].makeGui
    y = {Out.ar(~bus, Saw.ar(50, 0.1))} .play

    y.free

    // different source
    y = {Out.ar(~bus, SinOsc.ar(50, 0, 0.1))}.play
    s.freeAll
    (
    ~buf.free;
    ~bus.free;
    )
```

Figure 19.32
Rewriting with variable buffer length and GUI control.

Figure 19.33 shows the use of nonphasor read and write pointers (and hence the simultaneous application of buffer modulation and rewriting).

```
    // reading and writing with UGens other than Phasor
    (
    ~buf = Buffer.alloc(s, 1000);
    ~bus = Bus.audio(s, 1);
    )

    // control of write and read ugen frequencies with MouseX and // Mo
    useY
    // start fx
    (
    x = {
        var inSig = In.ar(~bus, 1);
        var read, write;
        write = BufWr.ar(inSig, ~buf, SinOsc.ar(MouseX.kr(3, 20),   0, B
    ufFrames.ir(~buf)));
```

```
    read = BufRd.ar(1, ~buf, SinOsc.ar(MouseY.kr(3, 20) *   [1, 1.0
1]).range(0, BufFrames.ir(~buf)));
    Limiter.ar(LPF.ar(read, 5000), 0.2)
}.play
)

y = {Out.ar(~bus, SinOsc.ar(100, 0, 0.1))} .play

s.freeAll
(
~buf.free;
~bus.free;
)
```

Figure 19.33
Buffer modulation and rewriting combined.

Feedback is an option to extend delay effects. (See figure 19.34.)

```
// Feedback of reading into writing
(
~buf = Buffer.alloc(s, 5000);
~bus = Bus.audio(s, 1);
)
// control of read rate with MouseX
// start fx silently
(
x = {
    var inSig = In.ar(~bus, 1);
    var sig, read, write;
    // this setup works nice with fixed feedback amplitude of // 0.9
9. . .
    sig = (LocalIn.ar(1) * 0.99) + inSig;
    //. . .and with write rate around 1
    write = BufWr.ar(sig, ~buf, Phasor.ar(0, MouseX.kr(0.99, 1.01),
0, BufFrames.ir(~buf)));
    read = BufRd.ar(1, ~buf, Phasor.ar(0, [1, 1.001], 0,   BufFrame
s.ir(~buf)));
    LocalOut.ar(read[0]);
    // LocalOut.ar(read[0].tanh); // check this

    Limiter.ar(LPF.ar(read, 5000), 0.2)
}.play
```

```
)

// start source
y = {Out.ar(~bus, SinOsc.ar(MouseY.kr(50, 500), 0, 0.1))} .play

// feedback continues for a long time after freeing the source
y.free
x.free;

(
~buf.free;
~bus.free;
)
```

Figure 19.34
Buffer rewriting with feedback.

Figure 19.35 demonstrates the possibility of overdubbing instead of plain rewriting.

```
// overdub, GUI control
// instead of plain rewriting, a mix of the current buffer // conte
nt and the in signal is written

(
~buf = Buffer.alloc(s, 5000);
~bus = Bus.audio(s, 1);
)

(
SynthDef(\rewrite_overdub, {|out, readRate = 0.8, writeRate = 1.4,
odubMix = 0.85, odubMode = 0,
   length = 0.2, cutoff = 5000, amp = 1|
   var inSig = In.ar(~bus, 1);
   var read_1, read_2, write, writePhase, odub;

   writePhase = Phasor.ar(0, writeRate, 0, BufFrames.ir(~buf) *   l
ength);

   // need to read the current buffer content before mixing:
   // strict overdubbing would have to take the floor of the // pha
se as BufWr doesn't round
   // make a distinction between floor and non-floor by // argument
odubMode
```

```
   read_1 = BufRd.ar(1, ~buf, [writePhase, writePhase.floor]);
   read_1 = Select.ar(odubMode, read_1);
   odub = inSig * (1—odubMix) + (read_1 * odubMix);

   write = BufWr.ar(odub, ~buf, writePhase);
   read_2 = BufRd.ar(1, ~buf, Phasor.ar(0, readRate * [1, 1.01],
 0, BufFrames.ir(~buf) * length));

   Out.ar(out, Limiter.ar(LPF.ar(read_2, cutoff), amp))
}, metadata: (
   specs: (
        readRate: [0.2, 2, \lin, 0, 0.8],
        writeRate: [0.2, 2, \lin, 0, 1.4],
        odubMix: [0, 1, \lin, 0, 0.85],
        odubMode: [0, 1, \lin, 1, 0],
        length: [0.01, 1, 5, 0, 0.2],
        cutoff: [50, 16000, \exp, 0, 5000],
        amp: [0, 0.5, \db, 0, 0.5]
   )
)
).add
)

// start fx with GUI before source
SynthDescLib.global[\rewrite_overdub].makeGui

// nice resonance and echo effects can occur with decaying // impul
ses
y = {Out.ar(~bus, Decay.ar(Impulse.ar(2), 0.1))} .play

s.freeAll

(
~buf.free;
~bus.free;
)

// check examples with different blockSizes
(
s.options.blockSize = 1;
s.reboot;
)
```

Figure 19.35
Buffer rewriting with overdubbing.

As audio data are read and written in blocks, the chosen block size can have consequences for the result of the procedure. In the example shown in figure 19.34, the block size has an additional impact; it also determines the minimum feedback delay time.

The presented buffer modulation and rewriting procedures offer a lot of possibilities not only by the variants in which they use `BufRd` and `BufWr`, but also because any source can be taken as buffer content, modulator, or for rewriting. This distinguishes them as extraordinarily versatile tools for explorative sound synthesis.

The `miSCellaneous_lib` quark extension[10] also contains classes suited for experimental synthesis, including single-sample feedback (`Fb1`, Mayer 2020), the audification of systems of ordinary differential equations (`Fb1_ODE`, Mayer 2021), and the sequencing of effects and effect graphs (`PbindFx`, Mayer 2019). Some classes provide alternative implementations for partially explored but still promising techniques like wavefolding (`SmoothFoldS` a.o.), wavesets (`ZeroXBufRd` / `TZeroXBufRd`), functional iteration synthesis (`GFIS`), and sieves (`Sieve` / `Psieve`).

The techniques presented in this chapter are only a fraction of the possibilities available in nonstandard synthesis, and we hope that readers will be inspired to explore variants of these processes, and further alternative synthesis methods, within SuperCollider.

Notes

1. https://www.youtube.com/watch?v=GtQdIYUtAHg.
2. https://www.youtube.com/watch?v=qlrs2Vorw2Y.
3. https://www.youtube.com/watch?v=tCRPUv8V22o.
4. https://arxiv.org/pdf/1112.1368.pdf.
5. https://schemawound.com/category/posts/tutorial/supercollider/.
6. https://www.researchcatalogue.net/view/921059/922509.
7. https://arxiv.org/pdf/1112.1368.pdf.
8. https://listarc.cal.bham.ac.uk/lists/sc-users/msg68907.html.
9. https://supercollider.github.io/workshops/2012/03/25/a-hands-on-look-at-some-exotic-techniques-for-sound-generation-and-processing-barcelona-202122042012.html.
10. https://github.com/dkmayer/miscellaneous_lib.

References

Berg, P. 1979. “Background and Foreground.” In *SSP. A Bi-parametric Approach to Sound Synthesis. Sonological Reports* (Vol. 5), pp 9-32. Utrecht, Netherlands: Institute of Sonology.

Berg, P. 2009. “Composing Sound Structures with Rules.” *Contemporary Music Review,* 28(1): 75–87.

Collins, N. 2008. “Errant Sound Synthesis.” *Proceedings of ICMC2008*, International Computer Music Conference, Belfast.

Collins, N. 2011. “Implementing Stochastic Synthesis for SuperCollider and iPhone.” *Proceedings of Xenakis International Symposium.*

Holtzman, S. R. 1978. “A Description of an Automatic Digital Sound Synthesis Instrument.” In *D.A.I. Research Report* (Vol. 59). Edinburgh: Department of Artificial Intelligence.

Koenig, G. M. 1970. “Project 2: A Programme for Musical Composition.” In *Electronic Music Reports* (Vol. 3). Utrecht, Netherlands: Institute of Sonology.

Luque, S. 2006. “Stochastic Synthesis: Origins and Extensions.” Master’s thesis, Institute of Sonology, The Hague, Netherlands.

Luque, S. 2009. “The Stochastic Synthesis of Iannis Xenakis.” *Leonardo Music Journal,* (19): 77–84.

Mayer, D. 2019. “PbindFx: An Interface for Sequencing Effect Graphs in the SuperCollider Audio Programming Language.” *Proceedings of the 14th International Audio Mostly Conference: A Journey in Sound*, pp. 287–291. https://doi.org/10.1145/3356590.3356639.

Mayer, D. 2020. “Fb1—An Interface for Single Sample Feedback and Feedforward in SuperCollider.” *Proceedings of International Conference on Sound and Music Computing (SMC2020)*, Turin, Italy, pp. 200–203. https://zenodo.org/record/3898757.

Mayer, D. 2021. “Fb1_ODE—An Interface for Synthesis and Sound Processing with Ordinary Differential Equations in SuperCollider.” *Proceedings of International Conference on Sound and Music Computing (SMC2021)*, Turin, Italy, pp. 154–158. https://zenodo.org/record/5038733.

Puckette, M. 2007. *The Theory and Techniques of Electronic Music*. Singapore: World Scientific Publishing.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

Smith, J. O. 1991. “Viewpoints on the History of Digital Synthesis.” In *Proceedings of the 1991 International Computer Music Conference*. San Francisco, International Computer Music Association.

Solomos, M. 2001. “The Unity of Xenakis’s Instrumental and Electroacoustic Music: The Case for ‘Brownian Movements.’ ” *Perspectives of New Music,* 39(1): 244–254.

Truax, B. 1973. “The Computer Composition—Sound Synthesis Programs POD4, POD5 & POD6.” In *Sonological Reports* (Vol. 3). Utrecht, Netherlands: Institute of Sonology.

Xenakis, I. 1992. *Formalized Music: Thought and Mathematics in Music,* rev. ed. Hillsdale, NY: Pendragon Press.

V PROJECTS AND PERSPECTIVES

20 Implementing New Language Syntax Using SuperCollider's Preprocessor

H. James Harkins

20.1 Introduction

SuperCollider (SC) supports custom language syntax, implemented within itself, by passing code text into a preprocessor function before running the code. The preprocessor is free to modify or even fully replace the input code, so long as the function's result is valid SuperCollider language code. This ability to use the language to redefine the language opens the door to alternate dialects, ranging from keyword replacement all the way up to pattern-definition languages suitable for live-coding improvisation. Parsing and manipulating syntactical units may require some sophistication, however; the purpose of this chapter is to introduce techniques to use the preprocessor in more complex and reliable ways.

The preprocessor is defined in the `Interpreter` class as a variable, `preProcessor`. Normally, `preProcessor` is nil; user code passes directly to the sclang compiler. If a `Function` is assigned to `preProcessor`, the function is evaluated with the user code as input. Its output should be a `String`, consisting of valid `sclang` syntax to execute. Between input and output, there are few limits (in principle) on the complexity of the processing that is possible: simple string substitution, extension of abbreviated syntax, or even compilation of complex expressions.[1]

To introduce the preprocessor, we will begin with the simplest usage: keyword substitution. From there, we will examine techniques to be aware of context in the input code and go on to apply these techniques to rearrange the order of expressions within dynamic strings. Finally, we will draw some insights from compiler design for more robust parsing and translation of custom dialects. In this discussion, I am drawing on ten years of experience as the developer of a live-coding pattern dialect, ddwChucklib-livecode[2] ("cll" for short), implemented entirely in SuperCollider. The design principles described in this chapter are based on cll's preprocessor; interested readers may access the repository to see how the approach scales up to a fully functioning performance dialect.

20.2 String Substitution

Imagine that, for whatever reason, we find ourselves writing `thisThread.clock` frequently. This may become tiresome, so we might want to be able to write a shorter keyword and let it be expanded upon evaluation.

A preprocessor for this scenario begins with a simple idea. Sclang's `String` class already implements a find-and-replace method, so the preprocessor can accept the string as an argument and use the `replace` method to remove the custom keyword `$clock` and replace it with the valid syntax `thisThread.clock`. This function is assigned to `this.preProcessor`, as in figure 20.1. (Note that `replace` returns the original string's contents if the search string is not found, so no special handling is required for normal code blocks.)

```
(
this.preProcessor = {|str|
   str.replace("$clock", "thisThread.clock");
};
)

// Usage:
TempoClock.sched(0, {("tempo is" + $clock.tempo).postln;   $c.postl
n;});
```

Figure 20.1
Simple find-replace keyword substitution

In the keywords to be replaced, it is advisable to avoid valid sclang syntax. We may use "$clock" because there is no standard sclang expression in which "$clock" would be allowed (outside of a quoted string or symbol). `$c.postln` is included in the example to show that `$c` is valid syntax for a character literal. `$cl…`, with multiple characters, is not valid. Therefore, we can replace `$clock` freely and be confident that the replacement will not be triggered accidentally. An alternate syntax, "#clock," should not be chosen, because it is valid in multiple-assignment syntax: e.g., `#clock, time, action = someArray`.[3]

"$clock" is a fixed keyword. To match keywords with variable components, or even more complex structures, regular expressions may be a good choice, using String's `findRegexp` or `findAllRegexp` methods. Figure 20.2 searches for a dollar sign followed by any keyword of at least two characters, and substitutes `thisThread.` in place of the dollar sign. This one preprocessor changes `$clock` into `thisThread.clock`, `$beats` into `thisThread.beats`, and so on, while character literals such as `$x` remain unchanged (because of the two-character minimum). Regular

expressions are a large part of the implementation of at least one SuperCollider live-coding dialect, Bacalao.[4]

```
(
this.preProcessor = {|str|
   str.findRegexp("\\$[a-z][a-zA-Z0-9]+").reverseDo {|match|
        var pos = match[0];
        str = str[0 .. pos-1] ++ "thisThread." ++ str[pos + 1
 ..];
   };
   str
};
)

// Usage:
"On %, current time is %\n".postf($clock, $beats);
```

Figure 20.2
Regular expressions for keyword searching.

Quickly, however, we run into a problem with the find-and-replace approach. The statement in figure 20.3 includes the search token both within and outside a string literal. Both are converted: the printed output is "thisThread.clock is a TempoClock." If we want the string literal to be untouched, printing the result "$clock is a TempoClock" instead, this is not possible with either figure 20.1 or 20.2. All occurrences will be replaced, regardless of context. The next step, then, is to develop techniques that are aware of the context of the expressions.

```
TempoClock.sched(0, {("$clock is" ++ $clock).postln});
```

Figure 20.3
Lack of context in string substitution.

20.3 Context-Aware Scanning

The problem in figure 20.3 is that there is a difference between characters within double- or single-quote string/symbol delimiters and undelimited expressions. The former case represents data; the latter, an operation. To tell the difference, we need to know when we are within string delimiters. If we scan forward through the code string, "string" versus "nonstring" is unambiguous. At the beginning of the code block, we are outside string delimiters. We proceed through the string, character by character, until

encountering a quote mark. Subsequent characters are string contents until the closing quote mark is reached.

We can think of this task in terms of two states: scanning code versus scanning literal data. The transitions between these states are well defined; in "code" state, a quote mark triggers the "string literal" state, and vice versa. In computer science terms, this is a "finite state machine": conditions encountered in the input cause the machine to switch among a finite set of states. The goal in this example, then, is to implement keyword substitution in "code" state, but not "string" state.

We will represent each scanning state in its own function. The transition from one state to another is simply a function call. Factoring into separate functions may seem unnecessary for such a simple state machine, but computer languages are usually deeply nested grammars. When a subelement has been fully parsed, it is necessary to resume the next higher level's element where it left off. Function calls and returns handle this requirement gracefully. If we begin with this approach now, we will be able to grow into much more complex expressions without requiring a fundamentally different code design.

We can speak of a scanner (parser) being "at" a position in the code string. This position needs to be available for every state function. The fact that we are stepping through the string by characters suggests a *stream*, which advances by calling `.next` on it. SuperCollider includes a convenient way to stream out the contents of a string: `CollStream` ("collection-stream").[5] We pass the stream object as an argument between the various states. The scanning functions consume characters from the source; when control returns to the next higher level, scanning continues as if it had not been interrupted.

Figure 20.4 reimplements the "$clock" preprocessor using a pair of functions, `scanCodeBlock` and `scanString`, to skip string literals. Here, `scanCodeBlock` is the main preprocessor and implements the state machine within its `while` loop. After reading a character from a stream, that character determines the next processing state: a dollar sign checks for the `$clock` token and replaces it; a double or single quote dispatches into `scanString`; and all other characters are copied directly to the output. This approach, a `while` loop with branching, has an advantage as a general format for a parsing function because we can simply add more branches to handle more syntactic elements. (Note that the branching need not be `switch`; `case` is equally valid, as it is a series of `if` statements.)

```
(
var scanCodeBlock = {|string|
    var stream = CollStream(string);
    var ch;
```

```
	var output = CollStream.new;

	while {
		ch = stream.next;
		ch.notNil
	} {

		switch(ch)
		 {$$} {
			if(string[stream.pos .. stream.pos + 4] == "clock") {
				stream.nextN(5); // consume "clock"
				output << "thisThread.clock"
			} {
				output << ch; // don't drop $ for char literals!
				// but, char literals also need to swallow the c
har
				ch = stream.next;
				if(ch.notNil) {
					output << ch
				}
			}
		}
		 {$"} {
			output << ch << scanString.value(stream, $") << ch;
		}
		{$'} {
			output << ch << scanString.value(stream, $') << ch;
		}
		// default case
		{output << ch};
	};
	output.collection;
};

var scanString = {|stream, delimiter|
	var ch;
	var string = String.new;
	while {
		ch = stream.next;
		ch.notNil and: {ch != delimiter}
	} {
		string = string.add(ch);
		// handle escaped characters
```

```
        if(ch == $\\) {
                ch = stream.next; // process next char, without   del
imiter check
                if(ch.notNil) {string = string.add(ch)};
        }
    };
    string // return scanned string back to caller
};

this.preProcessor = scanCodeBlock;
)

TempoClock.sched(0, {("$clock is" ++ $clock).postln});
// Sample output: $clock is a TempoClock
```

Figure 20.4
String replacement, with awareness of string-literal context.

When we take responsibility for character-by-character parsing, edge cases quickly emerge. For example, `scanString` should handle escaped characters: `"They shouted, \"Hello, World!\" in unison"` is a valid string, containing two quote marks; `Hello, World!` in the middle should remain in "string" state. Thus, the body of the while loop needs to "eat" any character following a backslash, excluding it from the end-quote check. A similar problem occurs with character literals: `$"` does not denote the start of a string. Thus, if the dollar-sign branch does not match "$clock," then it needs to consume one more character and protect it from branching to another state. (A remaining problem is that figure 20.4 does not handle comments. If an isolated quote mark occurs within a comment, then subsequent string-scanning will be out of sync with the actual code structure. To be entirely correct, then, we would need to add two more state functions: one to skip `/* delimited comments */` and another for `// single-line comments`.) Such cases are a general trap in preprocessor design: even seemingly simple requirements may require careful handling of subtleties. It is not safe to assume that input will always conform to initial expectations!

It is also necessary to pay close attention to the stream's position when entering and exiting a parsing function. For instance, upon entering `scanString`, should the next character to be pulled from the stream be the open quote mark or the character after the quote mark? If the next character is the open quote, then `scanString` can determine the delimiter by pulling the next character. But the calling function will have already consumed this character in order to determine that the next element is a string literal! So the caller would then have to `stream.rewind(1)` to make the quote mark available again. This programming interface would be more delicate because the caller needs to

manipulate the stream before calling the function. The solution taken in figure 20.4 is that `scanString` starts reading one character after the quote mark and receives the delimiter as an argument. This solution strikes me as less "sneaky" than the extra manipulation (rewind) on the code stream. In either case, however, the caller needs to be aware of the downstream function's requirements for stream position.

If all of this sounds delicate, that is because it is. It requires unambiguous design of the input syntax, careful attention to detail, and extensive debugging for the inevitable mistakes that will be made. But all the pitfalls do have solutions.

20.4 Reordering Syntactic Elements: Formatted String Literals

In the previous examples, the order of syntactic elements did not change: the sclang expression `thisThread.clock` is placed in the same location as the custom token `$clock` keyword. Let us consider a case where the order of code elements must be rearranged.

As an example, we will implement Python's concept of "formatted string literals," or f-strings, where expressions are embedded into a string literal. At runtime, `f"One plus one = {1 + 1}"` should evaluate `1 + 1` and substitute the result into the string, producing a new string: `"One plus one = 2"`. SuperCollider supports the runtime insertion of expression results into strings using the `format` method: we can write `"One plus one = %".format(1 + 1)` and get the same result. The difference is that, where Python intersperses expressions within the string text, SuperCollider requires all the expressions to be postfixed as arguments to the `format` method. A translator, then, needs to extract and save the code for the expressions so they can be written out at the end.

As in section 20.3, we will break the processing into smaller functions. In figure 20.5, the structure of `convertFStrings` matches that of figure 20.4's `scanCodeBlock`: a `while` loop pulling characters from a `CollStream`, using these to dispatch to the f-string handler or to bypass normal string literals. Here, `scanNormalString` is simply copied from figure 20.4.

```
(
var convertFStrings = {|code|
    var stream = CollStream(code);
    var out = CollStream.new;
    var ch, closeQuote;
    while {
        ch = stream.next;
        ch.notNil
    } {
```

```
        case
        {ch == $f and: {stream.peek == $"}} {
              stream.next; // swallow quote
              out << parseFString.(stream);
        }
        {ch == $" or: {ch == $'}} {
              out << ch << scanNormalString.(stream, ch) << ch;
        }
        {ch == $$} {
              // see above: $" is valid and must not trigger string
-literal state
              out << ch;
              ch = stream.next;
              out << ch;
              // and. . .this branch because $\" is a valid charact
er literal too
              if(ch == $\\) {
                    out << stream.next;
              };
        }
        {
              out << ch;
        };
   };
   out.collection
};

var scanNormalString = {|stream, delimiter|
   var ch;
   var string = String.new;
   while {
        ch = stream.next;
        ch.notNil and: {ch != delimiter}
   } {
        string = string.add(ch);
        if(ch == $\\) {
              ch = stream.next;
              if(ch.notNil) {string = string.add(ch)};
        }
   };
   string // return scanned string back to caller
};
```

```
var parseFString = {|stream|
   // assumes 'f" ' is already scanned; return: code converted to
 "str".format(. . .)
   var start = stream.pos;
   var list = List.new;
   var formatStr = String.new;
   var out, ch;
   // Stage 1: Extract expressions
   while {
        ch = stream.next;
        ch.notNil and: {ch != $"}
   } {
        switch(ch)
        {${} {
              parseOneExpression.(stream, list);
              formatStr = formatStr ++ "%";
        }
        {$\\} {
              formatStr = formatStr ++ ch ++ stream.next;
        }
        {formatStr = formatStr ++ ch};
   };
   // reached the end without a closing quote: syntax error
   if(ch.isNil) {
        Error("open-ended f-string: %".format(stream.collection
 [start .. start + 20])).throw;
   };
   // Stage 2: Code generation
   if(list.size >= 1) {// .format only if there's something to   fo
rmat
        out = CollStream.new;
        out <<< formatStr << ".format(";
        list.do {|expr, i|
              if(i > 0) {out << ", "};
              out << expr;
        };
        out << ")";
        out.collection
   } {
        // −1−we need the opening quote, and don't include .next ch
ar
```

```
        stream.collection[start − 1 .. stream.pos − 1]
    }
};

var parseOneExpression = {|stream, list|
    // assumes '{' is already scanned; result: item added to list
    var start = stream.pos;
    var out = CollStream.new;
    var ch;
    while {
        ch = stream.next;
        ch.notNil and: {ch != $}}
    } {
        case
        // check for nested f-string
        {ch == $f and: {stream.peek == $"}} {
            stream.next; // swallow quote
            out << parseFString.(stream);
        }
        // check for escape char
        {ch == $\\} {
            out << stream.next; // do not pass \ into expression
string
        }
        {out << ch}; // default case
    };
    if(ch.isNil) {
        Error("open-ended f-string argument: %".format(
            stream.collection[start .. start + 20]
        )).throw;
    };
    list.add(out.collection);
    stream
};

this.preProcessor = convertFStrings;
)

// Usage:
TempoClock.sched(0, {f"Current time is {thisThread.beats}".   postl
n});
// Sample output: Current time is 70.472959011
```

Figure 20.5
A preprocessor of f-string.

The heart of this preprocessor is `parseFStrings`. In this function, the input that we have is string text, with expressions in braces. The output that we need consists of two parts: first, a quoted string literal, where the brace-delimited expressions have been replaced by formatting placeholders; and second, an array containing the expressions. It is convenient that both the placeholder replacement and array construction happen in response to the same syntactic element (an opening brace). The open-brace `if` branch performs both of these; `parseOneExpression` adds the expression's text to the list, and `formatStr` appends the placeholder. When the end of the string is reached (closing quote), the remainder of the function generates and returns the output code.

This preprocessor is complex enough that it warrants a small testing suite. It is easy to think of correct use cases, but now we need to imagine how the preprocessor could go wrong. Here, we gain an advantage from designing the preprocessor in terms of a state machine. The weak points are likely to be in the transitions between states: entering or exiting a state inappropriately or failing to enter or exit a state when needed. What are the state transitions here? We enter an f-string from normal code, but not within a string literal: `postln(f"abc = {abc}")` is an f-string, while `postln("best of" + numTrials)` contains `f"` but is not an f-string. The latter case suggests that the test suite should verify that a string literal ending with f does not try to handle the subsequent code as an f-string. We exit an f-string when finding the closing quote, but we must skip over escaped quotes or quotes within braces. Expressions in braces should be processed only within f-strings, not anywhere else. Also, function braces used within f-string expressions should be possible to backslash-escape.

Based on such considerations, we may gather some test cases in table 20.1.

Table 20.1
Test cases of f-strings

Test Case	Input	Expected Output
An f-string with two simple expressions should pull both expressions into `format` with correct comma separation, and render the following code correctly.	`f"abc = {abc}, def = {def}"; xyz`	`"abc = %, def = %".format(abc, def); xyz`
A normal string ending with `f` should not trigger f-string logic. Also, escaped quotes should pass through correctly.	`"normal \"string\" ending with f".postln`	`"normal \"string\" ending with f".postln`
An f-quote within a symbol literal should not trigger f-string logic.	`'abcdef"xyz'`	`'abcdef"xyz'`
It should be possible to escape a closing brace within an f-string expression.	`f"random = {{1.0.rand \}.value}"`	`"random = %".format ({1.0.rand}.value)`

Test Case	Input	Expected Output
An f-string within an f-string expression should render as a nested `format` call.	`var abc = 10.rand; f"abc = {abc ++ f"nested string {xyz}"}".postln;`	`var abc = 10.rand; "abc = %".format(abc ++ "nested string %".format(xyz)).postln;`
An f-string expression missing its closing brace should throw an "Open-ended f-string expression" error.	`f"{abcxyz"`	(Check for the right `Error` to be thrown.)
An f-string missing its closing quote should throw an "Open-ended f-string" error.	`f"{abc}xyz; 123`	(Check for the right `Error` to be thrown.)

With clear requirements in hand, we can call the preprocessor function with testing strings and inspect the results, as in figure 20.6. The error-handling cases should use a `try` block, and verify that the `Error` object contains the expected text.

```
(

// get the f-string preprocessor function we defined previously
var convertFStrings = this.preProcessor;

c = "f\"abc = {abc}, def = {def}\"; xyz";
convertFStrings.(c).debug("basic case, 2 expressions");

c = "var abc = 10.rand; f\"abc = {abc ++ f\"nested string   {xyz}
\"}\".postln;";
convertFStrings.(c).debug("nested conversion");

c = "f\"random = {{1.0.rand \\}.value}\" ";
convertFStrings.(c).debug("escaped braces");

c = "\"normal \\\"string\\\" ending with f\".postln";
convertFStrings.(c).debug("escaped \"quotes\" and closing f");

c = " 'abcdef\"xyz' ";
convertFStrings.(c).debug("weird symbol");

try {
   convertFStrings.("f\"{abcxyz\" ")
} {|error|
   if(error.errorString.beginsWith("ERROR: open-ended f-string   ar
gument")) {
         "open-ended f-string argument detected OK".debug;
   } {
```

```
            error.throw;
        }
    };

    try {
        convertFStrings.("f\"{abc}xyz; 123")
    } {|error|
        if(error.errorString.beginsWith("ERROR: open-ended   f-string"))
    {
            "open-ended f-string detected OK".debug;
        } {
            error.throw;
        }
    };
    )
```

Figure 20.6
F-string preprocessor test suite.

To summarize, we are now extracting code content from parsing states and generating code based on the extracted strings. However, the code design has been procedural rather than object oriented. The next step, then, is to look at the benefits of parser objects over parser functions.

20.5 Using Objects to Understand Input Code

The examples so far have demonstrated procedures to translate user-defined syntax. Some types of syntax, however, require a deeper analysis of the semantics of, and relationships between, syntactic elements. In a procedural approach, information about these elements is lost when exiting a parsing function. A significant benefit of an object-oriented approach is that at the end of parsing, we have a network of objects representing the structure and meaning of the input code.

In particular, the design of many computer languages is nested, with larger syntactic units being made of smaller ones. A *tree* data structure captures this design perfectly. Figure 20.7 shows two examples with associated parsing trees: a standard SuperCollider expression and an f-string usage. Every element is represented, and the nested relationships explain each expression's structure. The f-string tree is simpler because, in this specific case, it is not necessary to understand the brace-delimited expressions only to retain the source code as a string. But the structure, in which an f-string node knows its formatting string and the associated expressions, explains the f-string in a way that the procedural code did not.

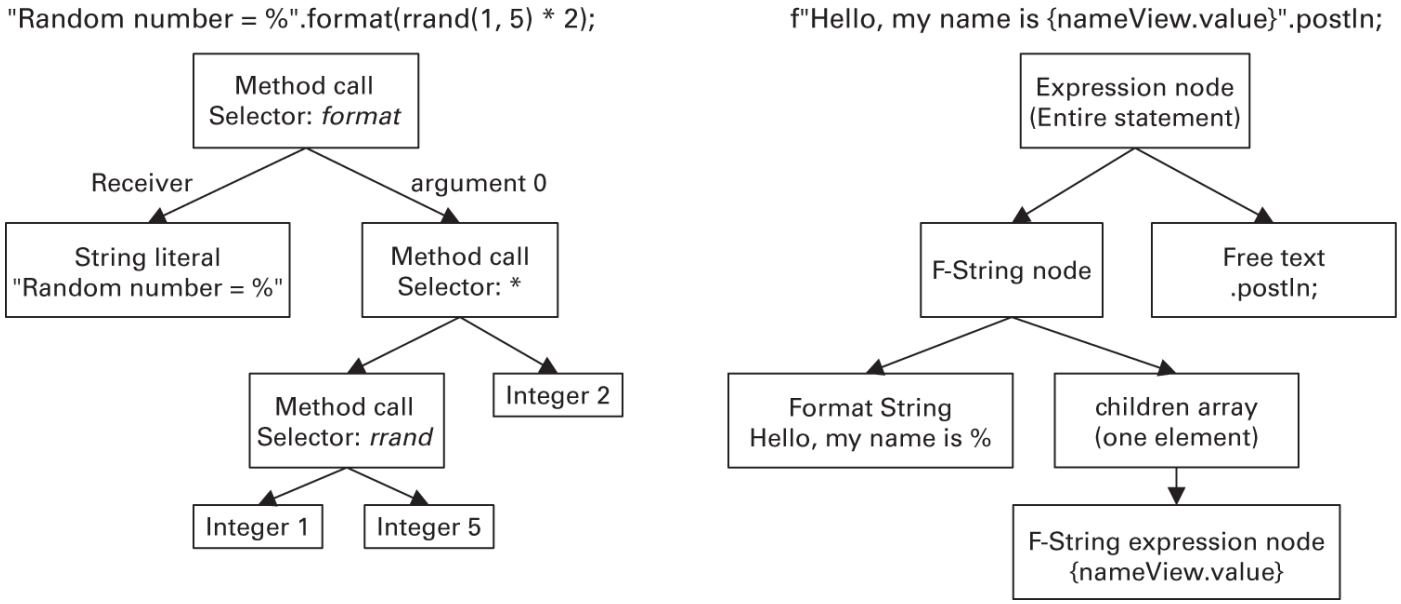


Figure 20.7
Tree structures derived from code statements.

A syntax tree may also generate output code based on the structure it represents. For instance, the "method call" object at the head of the left tree could generate code by following the following procedure:

1. Let the receiver object belonging to the method call write itself to the output.
2. Write a dot.
3. Write the method selector.
4. If there are arguments, write an open parenthesis. Then let each argument write itself to the output, with commas separating them. Last, append a closing parenthesis.

Each object is similarly responsible for rendering its own code. Thus, a complex code-generation requirement is broken down into much simpler components.

Turning now to implementation, every syntactic element needs to maintain certain common pieces of information: begin and end string positions, string contents, and links to parent and children nodes. Accordingly, we begin with an abstract class (figure 20.8) to declare these variables and some shared initialization methods. Of particular interest is the `parse` method. Every parsing operation needs to initialize the beginning position variable and to capture the ending position after scanning through the syntactic element. The abstract class defines `parse` to handle these common details, delegating the real work of parsing to another method, `doParse`. Subclasses should override `doParse` with their own parsing logic. (Note that the abstract class declares `doParse` to be the responsibility of a subclass. Abstract classes should not be used directly.)

```
SCBookAbstractParseNode {
   // these variables are general to parsing
   var <>begin, <>end, <>string, <>parentNode, <>children;

   *new {|parentNode|
        ^super.new.init(parentNode)
   }

   init {|argParent|
        parentNode = argParent;
        children = Array.new;
   }

   parse {|stream|
        begin = stream.pos;

        this.doParse(stream);

        if(end.isNil) {end = stream.pos - 1};
        if(string.isNil) {
              if(end >= begin) {
                   string = stream.collection[begin .. end]
              } {
                   string = String.new;
              };
        };
   }

   doParse {^this.subclassResponsibility(thisMethod)}

   /*** code generation template ***/

   streamCode {|stream| stream << string}

   /*** utility method ***/
   /* may be useful for all subclasses */
   skipSpaces {|stream|
        var ch;
        while {
              ch = stream.next;
              ch.notNil and: {ch.isSpace}
        };
        if(ch.notNil) {stream.rewind(1)};
```

```
    }
}
```

Figure 20.8
Abstract class for parse-tree nodes.

Space does not permit reproducing the entire set of f-string preprocessor classes; the entire working example may be found in the SuperCollider Book code repository. For illustration purposes, however, Figure 20.9 shows the parsing object for f-strings. The logic is very similar to `parseFString` in figure 20.5: a `while` loop, ending when the closing quote mark is reached, containing a branching statement (`switch`) to dispatch specific characters to other objects. For instance, the expressions themselves are delegated to `SCBookFExprNode(this).parse(stream)`; this is analogous to the function calls in the previous examples, but we also get new objects to be retained in the f-string node's collection of children. This is more general than the list of expression strings in the procedural version because every type of parsing object may have child nodes underneath it.

```
SCBookFStringNode : SCBookAbstractParseNode {
   var formatStr;
   doParse {|stream|
        var ch, new;
        formatStr = String.new;
        begin = begin - 1; // preserve opening quote in 'string'
        while {
              ch = stream.next;
              ch.notNil and: {ch != $"}
        } {
              switch(ch)
              {$\\} {
                    formatStr = formatStr ++ ch ++ stream.next;
              }
              {${} {
                    formatStr = formatStr ++ "%";
                    new = SCBookFExprNode(this).parse(stream);
                    children = children.add(new);
              }
              {formatStr = formatStr ++ ch};
        };
        if(ch.isNil) {
              Error("open-ended f-string at "
                    ++ stream.collection[begin .. begin + 10]
```

```
                ).throw;
            };
        }

        streamCode {|stream|
            stream <<< formatStr;
            if(children.size > 0) {
                stream << ".format(";
                children.do {|child, i|
                    if(i > 0) {stream << ", "};
                    child.streamCode(stream);
                };
                stream << ")";
            };
        }
    }
```

Figure 20.9
Parsing node for f-strings.

Code generation takes place in a second phase, implemented in each object's `streamCode` method. In the tree structure, each object generates some of its own syntax but delegates responsibility to the child objects to generate their own code. Formally, this is called *tree traversal*; if every object's `streamCode` method at some point calls `streamCode` on every one of its children, then it is guaranteed that the entire tree will be represented in the output string. The preprocessor, then, needs only to generate the tree and to call `streamCode` on the top-level object (figure 20.10).

```
(
this.preProcessor = {|str|
    var source = CollStream(str);
    var tree = SCBookExpressionNode.new.parse(source);
    var target = CollStream.new;
    tree.streamCode(target);
    target.collection
};
)
```

Figure 20.10
Preprocessor for f-string parsing classes.

20.6 Pattern Languages

One problem with the object-based f-string parser is that it nearly doubles the code size without really adding functionality. Let us now extend these concepts to a scenario where some details of the code output depend on the structure of the tree as a whole. Here, the tree structure greatly simplifies the implementation.

Let us consider a compact notation for note events, loosely inspired by the popular TidalCycles live-coding language.[6] We will write note events into bracketed expressions where, following the TidalCycles example, the duration of each member of a bracketed expression equally divides the time duration that the expression represents. An element may be an identifier naming the sound to produce, a spacer (period), or a bracketed subgroup; a subgroup's members equally divide the time allotted to the subgroup, from the next level higher. The outermost bracket group may be preceded by a number defining the number of beats within the group; if omitted, it will look to the default `TempoClock`'s `beatsPerBar`. Figure 20.11 shows some example patterns, with their associated rhythms.

A procedural approach will face a challenge here, in particular with regard to the subgroups. Take, for example, the fourth expression from figure 20.11. If we use parser functions, such as a hypothetical `scanBracketExpr` and `scanIdentifier`, then we would call first into `scanBracketExpr`, and next into `scanIdentifier` (which should be simple because identifiers do not need child elements). Then we reach an inner `scanBracketExpr`. For this bracketed expression to allot time to its children, it needs to know that it spans one beat. But this is not known until completing the entire outer-bracketed expression! Before `scanBracketExpr` can complete its work, it needs to know about syntax elements that have not been scanned yet: we cannot base present decisions on a future that has not yet arrived. A tree-based approach solves that problem neatly: first, collect objects representing all the elements; second, walk through the tree to apportion time based on complete knowledge of the tree's structure; third, render code.

1. [kick kick]

2. [kick . kick kick]

3. [kick rest kick kick]

4. [kick [. kick] . kick]

5. [kick . kick kick .]

6. 3[kick kick kick kick]

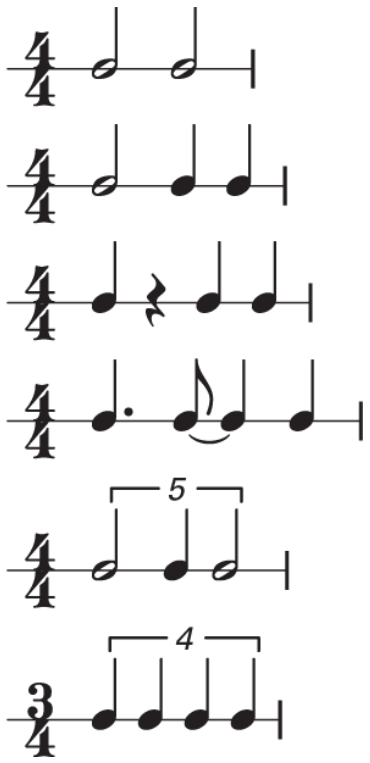


Figure 20.11
Mini-dialect rhythmic notation.

As the desired preprocessor behavior becomes more complex, it becomes more important to define the input and output formats carefully. Especially critical is that the input format needs to avoid syntactical ambiguity. In this example, identifiers are alphanumeric, a spacer is a dot, and a subgroup is introduced by a left bracket ([); every element is uniquely identifiable. More complex languages may need to use context to establish the meaning of similar syntax elements. For instance, an identifier in one context may represent a sound to be played, but in another context, it may be a function or pattern name.

The output format in this example needs to account for fixed sequences only. Thus, it may be as simple as an `Pseq` (sequential pattern) of `Events`. Each event should represent the sound identifier, the onset time within the bar, and the duration until the next `event`: for instance, the expression `[kick kick]` would produce the pattern `Pseq([(id: 'kick', time: 0.0, dur: 2.0), (id: 'kick', time: 2.0, dur: 2.0)], 1)`. Spacers will fold into event durations and will not be rendered into code. Upfront decisions about the output format affect the shape of `streamCode` methods.

The complete implementation spans 275 lines; for space reasons, it is available online but not printed herein. I will explain some key implementation details, however. As in the preceding section, we will need to define classes for each syntactic element: `SCBookNoteNode` for sound identifiers, `SCBookNumberNode` for the bracket-group duration, `SCBookSpacerNode` for spacers, `SCBookBracketNode` for a bracketed group

or subgroup, and `SCBookPatternNode` for the top-level container. These all derive, as before, from an abstract class, `SCBookAbstractPatternNode`. These elements are parsed using the same techniques discussed earlier in this chapter. Each class's `doParse` method pulls characters from a `CollStream` within a `while` loop, and each character either belongs to the element being parsed or dispatches to a different element. Code generation likewise follows the pattern established above: we pass an output `CollStream` into `streamCode` methods, so each element is responsible for generating its own code.

As noted briefly earlier, in between parsing and code generation, this example needs to divide up time. For this purpose, the abstract class adds two new variables, `time` and `dur`.[7] Then, each element implements a method, `setTime`, taking as arguments this element's onset time and duration. Note and spacer elements do not subdivide, so they simply accept the duration as given. Bracket groups keep an array of children; the duration of each child element is, then, `dur / children.size`. From this, the onset time of each child can be calculated and passed down. Any bracketed subgroups among these children will in turn call `setTime` on their children; thus, one call to `setTime` at the top level walks through the entire tree.[8]

The classes discussed so far only handle bracketed groups, not the standard sclang code containing them. The preprocessor function (figure 20.12) needs to locate the bracketed groups, pass them to the compiler classes, and copy the rest of the code to the output. As before, we need to distinguish these bracketed groups from normal sclang syntax. I chose to mark them with a tilde. In normal sclang syntax, a tilde introduces an environment variable, so the tilde may precede only a lowercase letter. Bracketed pattern groups begin with either a left bracket or a duration number, so it is impossible to confuse this with standard syntax. The approach taken in the online example is to use a regular expression search (`findAllRegexp`) to find the character index of every bracketed group.[9] Then, for each group, we copy the freestanding code leading up to that point and then use the parsing objects to translate the bracketed group and append this to the output. Any remaining freestanding code is copied after the last group.

```
(
thisProcess.interpreter.preProcessor = {|str|
   var places = str.findAllRegexp("~[0-9\\[]");
   var inStream, outStream, parsed;
   if(places.isEmpty) {
        str // nothing to do, return original
   } {
        inStream = CollStream(str);
        outStream = CollStream.new;
        try {
```

```
                places.do {|index, i|
                        outStream << inStream.nextN(index - inStream.po
s);
                        // .next - parser isn't interested in ~
                        inStream.next;
                        parsed = SCBookPatternNode.new.parse(inStream);
                        parsed.setTime
                        .streamCode(outStream);
                };
                // copy tail of string
                outStream << str[inStream.pos ..];
        } {|err|
                // no stack trace for parser errors
                err.errorString.postln;
                outStream = CollStream.new;
        };
        outStream.collection
    };
};
)
```

Figure 20.12
Preprocessor for the bracket-group dialect.

The preprocessor makes no assumptions about the way that an audible pattern would use the encoded IDs. Figure 20.13 shows a simple usage example, in which the event's identifier is copied to the instrument key, choosing a `SynthDef` to play. It would be possible to add ID-translation functions into the example dialect, or other numeric parameters, but this is beyond the scope of this chapter.

```
(
SynthDef(\sn, {|out, ffreq = 172, rq = 0.15, amp = 0.1|
   var feg = EnvGen.kr(Env([4, 1], [0.32], \exp));
   var eg = EnvGen.kr(Env.perc(0.01, 0.9), doneAction: 2);
   var src = Array.fill(2, {PinkNoise.ar});
   var sig = BPF.ar(src, (ffreq * feg).clip(20, 20000), rq) *   (eg
* amp / rq);
   Out.ar(out, sig);
}).add;

SynthDef(\k, {|out, basefreq = 55, bendAmt = 4, bendTime = 0.07,
 decay = 0.3, amp = 0.1|
```

```
    var feg = EnvGen.kr(Env([bendAmt, 1], [bendTime], \exp));
    var eg = EnvGen.kr(Env.perc(0.01, decay), doneAction: 2);
    var osc = (SinOsc.ar(basefreq * feg) * 3).tanh;
    Out.ar(out, (osc * eg * amp).dup);
}).add;

f = Pbind(\instrument, Pkey(\id));
)

TempoClock.tempo = 136/60;

Pdef(\d).quant = -1; Pdef(\dplay).quant = -1;

Pdef(\d, Pn(~4[k sn [. k] sn], inf));

// to simplify live coding, we factor the id-translation
// into a separate Pdef
Pdef(\dplay, Pchain(f, Pdef(\d))).play;

Pdef(\d, Pn(~4[k [sn . . sn] [. k] sn], inf));

Pdef(\d, Pn(~4[k [sn . . sn] [. sn k .] sn], inf));

Pdef(\dplay).stop;
```

Figure 20.13
Usage example of the pattern preprocessor.

The design of this pattern language is a simplified version of my own cll dialect; cll uses single-character identifiers and maps them onto user-defined parameters; the resulting events contain user-specified values rather than the identifiers given in the input. Cll also implements a suite of generator objects embedded into the pattern strings in the format `\generatorName(argument list)`. The generator parser is more complex than the examples from this chapter, but it follows the same structure: it distinguishes argument types based on character sequences at the current input position and dispatches to other parsing classes for each child argument. It is not possible to identify every argument type based on single characters; instead, I use `String`'s `findRegexpAt` method to match more complex character sequences at the current stream position and dispatch accordingly. I am omitting many details; interested readers may consult the cll repository to learn more. The main point to note is that the cll pattern parser is different in scope and complexity, but not in structure, from the design presented here.

20.7 Conclusion

When implementing a dialect using the SuperCollider preprocessor, one's first idea may be to use string searching and substitution. The limitations of this approach may not be immediately apparent, but as the dialect becomes more complex, it may become necessary to distinguish syntactic elements based on context and to handle recursive structures in ways that expose the limitations of regular expressions. Designing the parser around a state machine by building a tree of objects neatly handles both problems.

A recent, exciting development is the emergence of Sparkler,[10] a sclang grammar for the ANTLR[11] parser generator. A parser generator writes the parser code for you based on an abstract grammar specification. The grammar defines atomic keywords and combinations of keywords; for instance, a dot-syntax method call in sclang is specified as `expr DOT name PAREN_OPEN argList keyArgList? PAREN_CLOSE`,[12] where each of the terms was defined previously. Note the absence of fiddly, per-character logic in the grammar definition; this becomes the responsibility of the generator! Currently, the project does not support sclang preprocessor usage. Preprocessor support would require the ability to generate sclang code to perform the parsing by adding a module, written in Java, to ANTLR's code base. A custom dialect would then be defined as an extension to Sparkler's sclang grammar. Once both of these are done, it should be dramatically simpler to extend the sclang syntax.[13]

In the meantime, the techniques laid out in this chapter might help users to design more complex, more useful preprocessors. Admittedly, code parsing remains a delicate operation, fraught with subtle details and opportunities for bugs. Readers should be prepared to undertake extensive debugging and testing for a complex preprocessor. However, the parser is likely to be the most stable component in a working dialect. Once debugged, it should require fewer changes than any other part of the system, meaning that attention paid to good design in the early stages will pay off for years to come.

Notes

1. A practical limit may be processing time. If a preprocessor takes more than 20–30 ms to complete its task, the user experience is likely to be sluggish, which may interfere with the timing of musical threads. It would take an extremely complex preprocessor to perform this poorly, however.

2. H. James Harkins, *ddwChucklib-livecode*. SuperCollider quark extension. https://github.com/jamshark70/ddwChucklib-livecode, accessed August 31, 2022.

3. Multiple-assignment syntax is documented in the Assignment Statements Help file, http://doc.sccode.org/Reference/Assignment.html#Multiple%20Assignment, accessed August 31, 2022.

4. Glen Fraser, *Bacalao*. SuperCollider quark extension, https://github.com/totalgee/bacalao, accessed September 4, 2022.

5. `CollStream` imitates the interface of `File`. (Reading file contents from a disk is, in fact, a streaming operation; `getChar` accesses the next character in the file.) The difference is that `File` requires a file on disk, while `CollStream` reads from or writes to a collection in memory.
6. Alex McLean et al. *TidalCycles*. Documentation at https://tidalcycles.org/docs/, accessed September 3, 2022.
7. To accommodate the new variables, I have changed the name of the abstract class. This is primarily to facilitate the installation of both sets of example classes side by side.
8. This discussion omits some details about the handling of spacer elements' time values. Because spacers do not render into the output code, their durations must be added into the immediately preceding note. This detail is specific to the example dialect and does not apply in general to all preprocessor usages; as it is not central to the chapter topic, I mention it in passing and leave it to readers to examine the `setTime` implementations on their own.
9. Note that this example does not implement context awareness, as in section 20.3.
10. Lucille Nihlen, *Sparkler: A SuperCollider parser library.* https://github.com/hadron-sclang/sparkler, accessed September 12, 2022.
11. Terence Parr, *ANTLR: ANother Tool for Language Recognition,* https://www.antlr.org/, accessed September 12, 2022.
12. Lucille Nihlen, *Sparkler: A SuperCollider parser library,* SCParser.g4 (SuperCollider language grammar), line 189. https://github.com/hadron-sclang/sparkler/blob/8f90d49a80d1543ca7be8611a37e364e4456e3fb/grammar/SCParser.g4#L189, accessed September 12, 2022.
13. Additionally, in 2023, Stefaan Himpe released the Scparco Quark, implementing parser combinators. The techniques in this chapter are similar to parser combinators, but they expose the full character-by-character logic. Scparco packages these techniques into easier-to-use objects. Stefan Himpe, *Scparco Quark*. https://github.com/shimpe/scparco, accessed September 24, 2023.

21 Interface Investigations

Thor Magnusson

21.1 Introduction

An *interface* is a field of abstractions where two systems interact with one another. We typically use this word to refer to the locus where a human and a machine communicate. The interface can be as simple as an "on/off" button or it can be multimodal: a mixture of types of hardware used to input commands into a system that responds through the use of screens, speakers, motors, and haptic feedback (Jensenius & Lyons, 2017). The computer is a metamachine with no *natural* interface, unlike physical machinery, where gears, buttons, and wheels are natural extensions of the mechanism itself. This fact is problematized when the computer is used for music, as we have innumerable arbitrary ways of representing an interface to the audio system of the computer. It could be anything from a simple "play" button to a custom-written class that encapsulates the digital signal processing of an audio unit generator (UGen). The questions here are concerned with purpose (Magnusson, 2010): What intentional bandwidth do we—as software designers—give to the users of our system? What degree of control do we provide, and which interfaces do we present as affordances of the system, such that the cognitive processes of the users can be reflected in the machine signal?

21.2 The Machine as Musical Instrument and Tool for Thought

The more control parameters an acoustic instrument has, the more difficult it is to master. The sophistication of an instrument entails particulars of fine control designed through ages of evolution. Consider the difference in the learning curve of the kalimba or kazoo compared to the violin or the piano (Jordà, 1995). When designing a musical system on the *computer*, we are obviously concerned with the musical parameters of the composition or the tool, but we also have to decide which ones to make controllable by users through some interface or other. In creating the interface, we decide upon the abstraction level and intensity of the system. Often, it is quite arbitrary which parameters are made controllable, as such decisions could depend upon the piece, the hardware, or the specific group of people for which the system is made. A problem arises when the situation changes and the designer or the user of the system decides to

change its internal variables. Not only will the piece be different, but the hardware will behave differently, causing regression of performance in the trained player and often general distress in the user group.

The problems of human–computer interaction (HCI) in computer music are deep and wide (Holland et al., 2013). They are partly exemplified in the highly interesting field of research currently centered on the New Interfaces for Musical Expression Conference (Refsum & Lyons, 2017; see also www.nime.org). As much research in the field shows, the prominent issues with digital instruments tend to be those of the unnatural mappings between gesture and sound; dislocation of sound source and instrument; and the dynamic nature of the instruments. The megalomaniac's dream musical interface might be one in which all human gesture could be translated to music. It would consist of some amazing tactile, motion-capturing, haptic feedback device that would map movement to sound. The problem with this notion is that of bandwidth. Making full use of the capabilities of this interface would require a dedication to embodied practice that is rarely found in the field of computer music. Research has shown that people like constraints and limited scope (Magnusson, 2019). Designing musical instruments is akin to designing a game. We are therefore forced to face the dynamic nature of these tools and how their design is essentially a process of defining constraints.

What are we creating when we design a musical system on a computer? We hardly want to limit ourselves by imitating the world of acoustic instruments. The history of computer music shows that its strength comes from the unique qualities of the computer as a fast and general number cruncher. As opposed to humans, computers excel at calculations, complex pattern recognition, analysis, and creativity from generative rules. The unique strength and innovative power of the computer in music lie elsewhere than in simulations. It can be found in its nature as an "epistemic tool" (Magnusson, 2009): a platform on which we can think about music—and think about ourselves thinking about music—due to its logical and self-referential nature. An environment like SuperCollider (SC), in its own way, is a system of thought in which we can think about music (McCartney, 1996). It is a perfect example of a musical environment in which musicians can externalize their thoughts and sketch the creation of musical systems that could become compositions (deterministic or generative), coplayers (intelligent and adaptive), or instruments (for live or studio use).

21.3 Introduction to ixiQuarks

The ixiQuarks were the result of my decision, having used SuperCollider for a long time, to write a modular system to produce patches for specific performances, on the one hand, and more complete instruments, on the other. The problem was that the

patches didn't integrate well: each used its own buffer mechanism, effects, bus routing, and timers. The ixiQuarks are a way of modularizing work patterns through custom-built tools, and they include buffer pools that take care of server buffer allocations, effect tools that include the most common effects (such as reverb, delays, filtering, etc.), and instruments that serve as pattern generators that allow the automation of certain processes while the user is focusing on something else. The ixiQuarks are well integrated with SuperCollider. One could, for instance, write a Pattern in a live-performance situation and route it out through a prebuilt ixiQuark reverb effect in a matter of a few seconds. Many of the instruments even contain code windows as part of their graphical user interfaces (GUIs). This makes their output more flexible, as the performers can write their own synthesis codes or even use the interface to communicate with other applications through protocols such as OSC (Open Sound Control) or Musical Instrument Digital Interface (MIDI). The ixiQuarks use a set of GUI views called *ixiViews*. Both of these code libraries exist in the SuperCollider Quark repository. (See figure 21.1.)

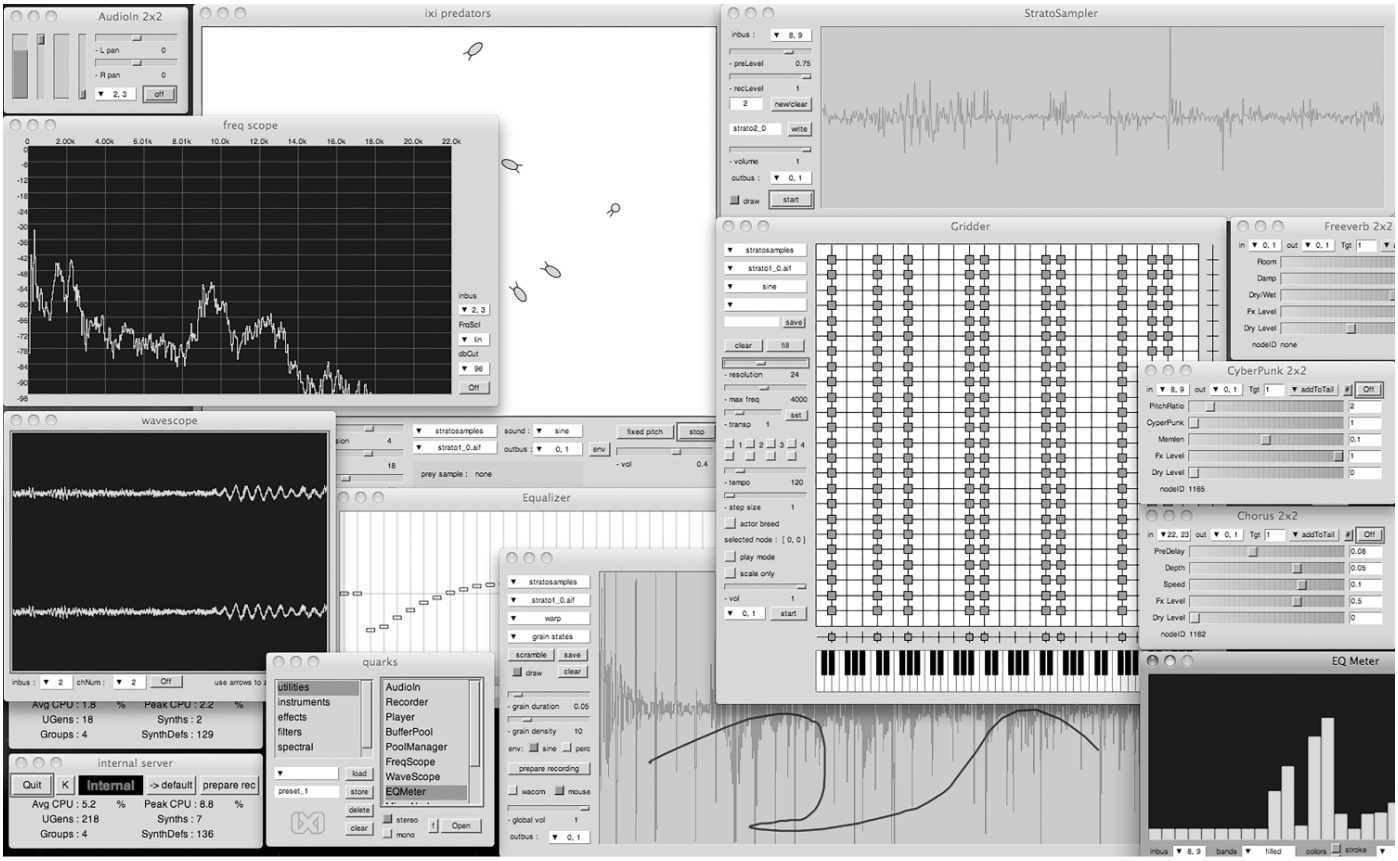


Figure 21.1
A screenshot of ixiQuarks.

What interests me in building musical instruments are the conceptual engines we create when we program our tools (Magnusson, 2007). From what ground do we start, and where do we end? How does the environment lead us? For me, SuperCollider is the ideal environment by far: it has a fantastic sound engine and a beautiful, object-oriented interpreted language; it is designed for music; it is open source; it has a good GUI tool kit (where for ixi, Pen, UserView, and Image are the most important classes apart from the standard views); and last, but not least, it has a great developer and user community. That said, as in any environment, SuperCollider has its design strategies and vision of what music and musical practice are. It is not a neutral blank slate of freedom, although it is arguably freer than most musical tools.

21.4 Case Studies: Designing the Touch Surfaces of Our Systems

When musicians work with code as artistic material, they often tend to explore the expressive potential of an environment through a process of bottom-up exploration. Contrary to computer scientists, who frequently design software from a top-down approach before coding it, musicians who code typically explore the complex relationships between different code elements, unit generators, and the interface functionality of the environment. Instead of seeing the programming language as a tool for imperative instructions, they use it as artistic material that can be formed like clay to sculpt works of art. Often the environment leads or inspires musicians, whether consciously or not, onto paths that they would not take otherwise. Let us now look at a few case studies that exemplify how a tool can imply creative leads.

21.4.1 Study 1: Mapping the Frequency and the Amplitude

Suppose that we want to design a simple synth with ten harmonics and control its amplitude and frequency. We create the synth definition `\simpleSynth` (see figure 21.2), which has a variable called `harmonics` so that we can set *at synthdef compile time* how many harmonics we want to have in our synth. This cannot be changed either in real time or when synths using this graph are instantiated. The `Mix.fill` mixes our 10 oscillators into a mono signal. Each oscillator has a frequency that is a harmonic (i.e., an integer multiple) of the fundamental. We divide the amplitude by the harmonic number so the higher frequencies are lower in amplitude than the lower ones. (If we wanted the same amplitude in all of them, we could simply use the `Blip` UGen.)

```
(
SynthDef(\simpleSynth, {|freq, amp|
   var signal, harmonics;
   harmonics = 16;
   signal = Mix.fill(harmonics, {|i|
```

```
                    SinOsc.ar(freq*(i+1), 1.0.rand, amp * harmonics
.reciprocal/(i+1))
                }));
    Out.ar(0, signal ! 2);
}).send(s)
)

// A line of code testing the synth definition that we created
Synth(\simpleSynth, [\freq, 440, \amp, 1])
```

Figure 21.2
A `Synth` definition with 10 harmonics.

We have created two inputs into the system of the `simpleSynth`. Its interface has two control parameters: frequency and amplitude. As these parameters can be changed in real time, we need to figure out a way to control them. Here, practicalities and intentions come into consideration. What are we designing, and whom are we designing it for? Is this for one-time use or wide distribution?

The most typical way to solve this is to use a slider (see figures 21.3 and 21.4).

```
(
var synth, win;
// we initialize the synth
synth = Synth(\simpleSynth, [\freq, 100, \amp, 0]);
// specify the GUI window
win = Window.new("simpleSynth", Rect(100,100, 230, 90), false);
// and place the frequency and amplitude sliders in the window
StaticText.new(win, Rect(10,10, 160, 20)).font_(GUI.font.new   ("He
lvetica", 9)).string_("freq");
Slider.new(win, Rect(40,10, 160, 24))
    .action_({|sl| synth.set(\freq, [100, 2000, \exp].asSpec   .map
(sl.value))});
StaticText.new(win, Rect(10,46, 160, 20)).font_(GUI.font   .new("He
lvetica", 9)).string_("amp");
Slider.new(win, Rect(40,46, 160, 24))
    .action_({|sl| synth.set(\amp, [0, 1.0, \amp].asSpec.map   (sl.v
alue))});
win.onClose_({synth.free}).front; // we add a "onClose"   message t
o the window and "front" it.
)
```

Figure 21.3
Code with horizontal sliders to control the frequency and amplitude of our synth.

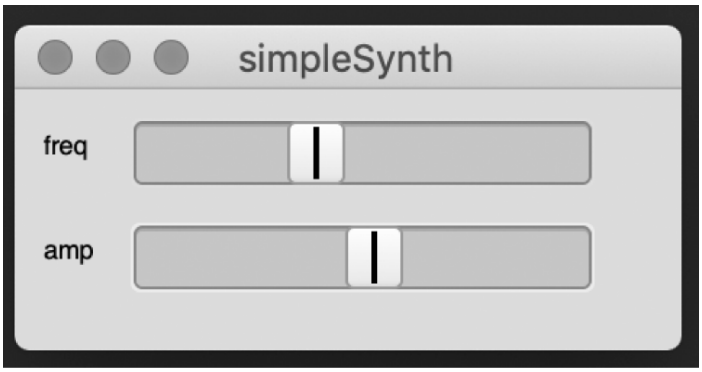


Figure 21.4
A screenshot of the GUI with horizontal sliders.

We have now plugged the sliders into the 2-dimensional interface of the synth. But we have done so without much thought. Why are the sliders horizontal? If we look at the description of the synth above, we read sentences such as "We divide the amplitude by the harmonic number such that the higher frequencies are lower in amplitude than the ones below." We use the *spatial* descriptors of high/low frequency and high/low amplitude—yet we create sliders that go from left to right. Which metaphors are we working with here: musical metaphors, in which our language and musical notation encourage us to see pitch and amplitude as up/down; or mathematical/scientific metaphors, in which negative numbers are on the left and positive numbers on the right? In fact, the GUI could just as well look as in figures 21.5 and 21.6.

```
(
var synth, win;
synth = Synth(\simpleSynth, [\freq, 100, \amp, 0]);
win = Window.new(" ", Rect(100, 100, 94, 200), false);
StaticText.new(win, Rect(20, 170, 160, 20)).font_(GUI.font   .new
("Helvetica", 9)).string_("freq");
Slider.new(win, Rect(10, 10, 30, 160))
   .action_({|sl| synth.set(\freq, [100, 2000, \exp].asSpec   .map
(sl.value)})};
StaticText.new(win, Rect(60, 170, 160, 20)).font_(GUI.font   .new
("Helvetica", 9)).string_("amp");
Slider.new(win, Rect(50, 10, 30, 160))
   .action_({|sl| synth.set(\amp, [0, 1.0, \amp].asSpec.map   (sl.v
alue)})};
win.onClose_({synth.free}).front; // we add a "onClose"   // messag
e to the window and "front" it.
)
```

Figure 21.5
Code with vertical sliders to control the frequency and amplitude of our synth.

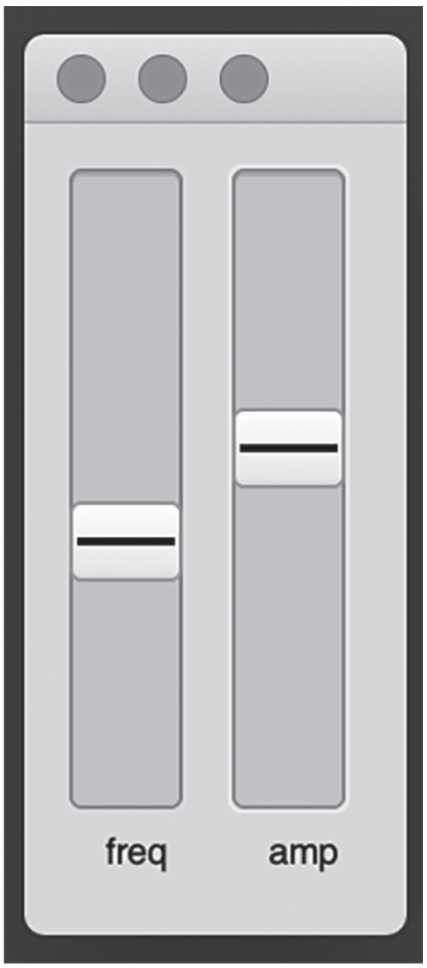


Figure 21.6
A screenshot of the GUI with vertical sliders.

21.4.2 Study 2: The GUI View as Inspiration and the ixiViews Quarks

Inspiration can come from anywhere. When working with GUI views, for instance, one often stumbles onto strange properties that can be used as a source for further work. Therefore, it makes sense to write GUI views that are as open and free as possible in terms of usage. A good example is `EnvelopeView`. Figure 21.7 presents a simple patch that came out of testing what happens when one connects all nodes in an envelope view.

```
(
var nNodes, envView, startStop, myWait, timeSlider, mouseTracker;
var xLoc, yLoc, mousedown = false;
var randLoc = 0.12.rand;

SynthDef(\irritia, {arg out=0, gate=1, freq=440, pan=0.0;
```

```
   Out.ar(out, Pan2.ar(LFSaw.ar(freq,0.4,0.05) * EnvGen.kr(Env .sin
e, gate, doneAction:2), pan))
}).add;

nNodes = 10;
myWait = 0.033;

w = Window.new("irritia", Rect(200 , 450, 400, 400)).front;

envView = EnvelopeView.new(w, Rect(20, 20, 355, 300))
              .thumbHeight_(6.0)
              .thumbWidth_(6.0)
              .fillColor_(Color.grey)
              .background_(Color.white)
              .drawLines_(true)
              .selectionColor_(Color.red)
              .drawRects_(true)
              .resize_(5) // can be resized and stretched
              .value_([{1.0.rand}!nNodes, {1.0.rand}!nNodes]);

// connect all the nodes in the envelope view to each other
nNodes.do({arg i; envView.connect(i, {|j|j}!nNodes);});

// create a little interaction where mouseactions affect the // act
ivity
UserView.new(w, Rect(20, 20, 355, 300))
   .mouseDownAction_({|view, x, y| mousedown = true;   xLoc = x/35
5; yLoc = (-1+(y/300)).abs;})
   .mouseMoveAction_({|view, x, y| xLoc = x/355;   yLoc = (-1+(y/30
0)).abs;})
   .mouseUpAction_({mousedown = false});

r = Routine({
   inf.do({|i|
        envView.selectIndex(envView.size.rand);
        if(mousedown.not, {
             0.05.coin.if({
                  0.5.coin.if({
                       myWait = rrand(0.028, 0.042);
                       xLoc = 1.0.rand;
                       yLoc = 1.0.rand;
                  });
                  randLoc = 0.12.rand2;
```

```
                });
                xLoc = envView.x+rand2(randLoc);
                yLoc = envView.y+rand2(randLoc);
            }, {
                xLoc = (xLoc + envView.x+rand2(0.1.rand))/2;
                yLoc = (yLoc + envView.y+rand2(0.1.rand))/2;
            });
            envView.x_(xLoc);
            envView.y_(yLoc);
            Synth(\irritia, [\freq, (yLoc*200)+50, \pan, (xLoc*2)-1]);
            myWait.wait;
        });
    }).play(AppClock);

    w.onClose_({r.stop});

    )
```

Figure 21.7
Irritia—A stochastic patch playing with the envelope view; the mouse can be used to interact with the patch.

The views of SuperCollider are basic and powerful, but there are limits to what can be done with them. This is where the `Pen` and the `UserView` classes can be useful, as they allow one to create custom GUI views. The ixiViews are examples of such custom views that afford functionality that the "native" GUI views of SuperCollider do not. The ixiViews are not the same as the ixiQuarks. They are used *in* ixiQuarks, but they are designed as general-purpose views to be used in various situations, so they are not as eccentric and constrained as the ixiQuarks. Figures 21.8–21.10 show screenshots of the ixiViews (`ParaSpace`, `Grid`, `BoxGrid`, and `MIDIKeyboard`) in real-world usage situations.

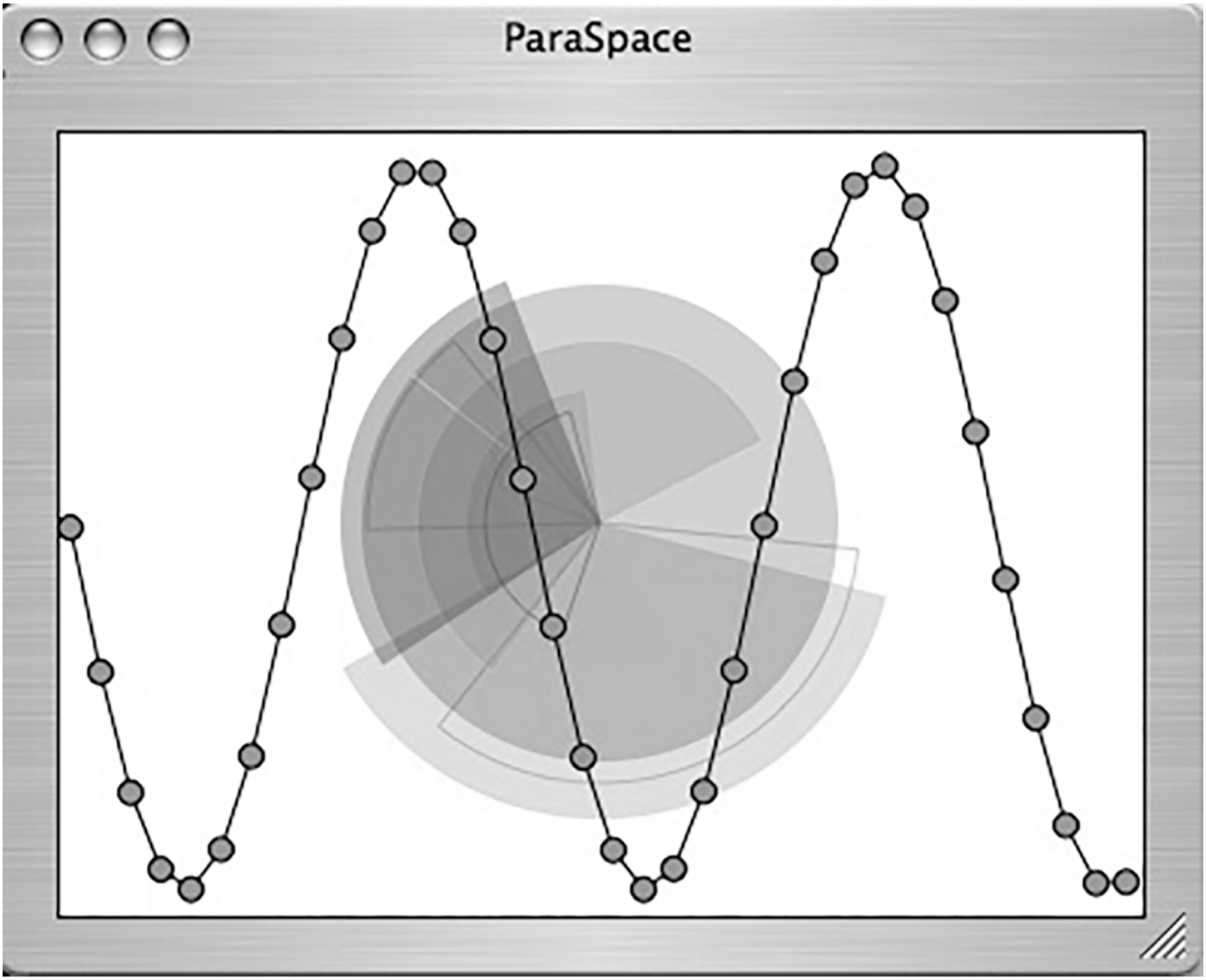


Figure 21.8
A screenshot of ParaSpace.

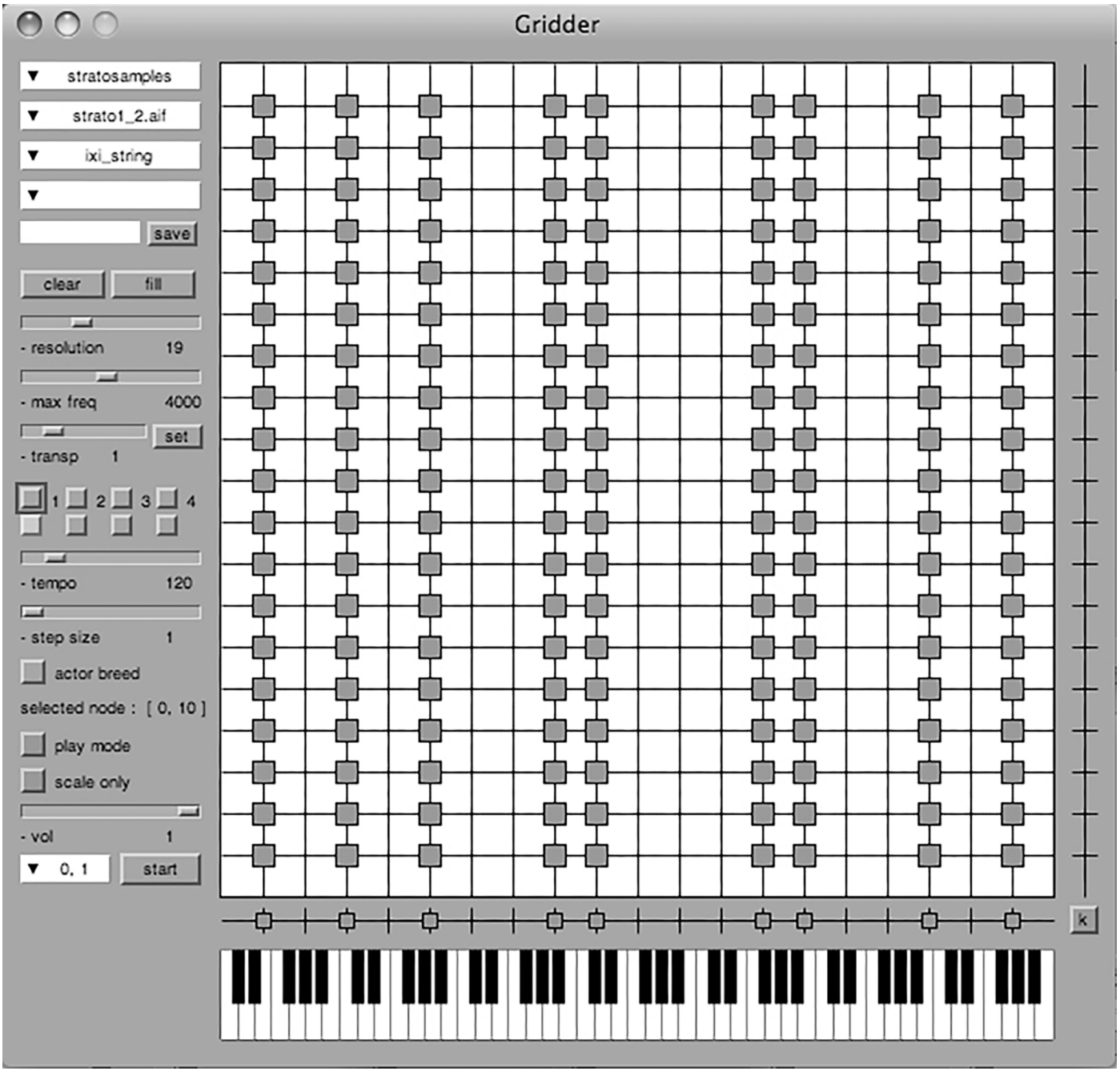


Figure 21.9
In the Gridder, we see how the `Grid` view is used to map microtonal scales (the horizontal columns) to the octaves (the vertical rows). One can then play the Gridder with the mouse or a pen tablet as if the lines of active nodes were strings. Agents can be set to move in the system and trigger automatic performance. We also see how the `MIDIKeyboard` view is used to show the pitch of the node on the grid. (It is gray if it fits the equal tempered scale, but red if it is a microtone.)

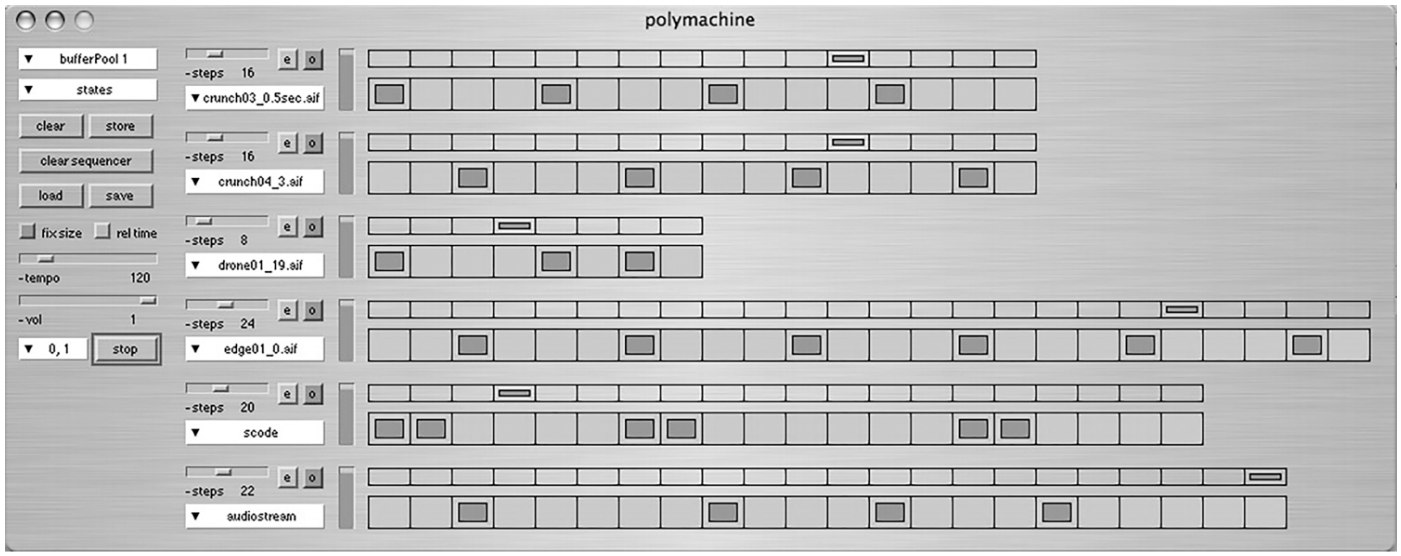


Figure 21.10
In the PolyMachine, the `BoxGrid` is used for polyrhythmic step sequencers. It can have any number of columns and rows, but here, it has only one row for each track and a row above it, for indicating which step the clock is at.

The existence of views such as ixiViews can encourage the production of alternative screen-based interfaces that encourage experimentation and improvisation. A good example is the Shooting Scales project, a collaboration between Shinji Kanki and myself. Here, six virtuoso pianists from the Piano NYT group are given game pads connected to a USB hub read with the HID class of SuperCollider. Each player has a keyboard (represented by the `MIDIKeyboard` class) and can set up scales, chords, and arpeggios with the game pad. Their actions are then generalized in the GUI with a larger keyboard. In a performance, that keyboard is "connected" to a Disklavier (a digital player piano) through MIDI. Thus, it is a 12-handed piece that is played by virtuosi pianists using an interface to the instrument that they may not be as accustomed to as the tactile keys of the physical keyboard. As SuperCollider supports networked communication, the Disklavier can be played remotely through the Internet. (See figure 21.11.)

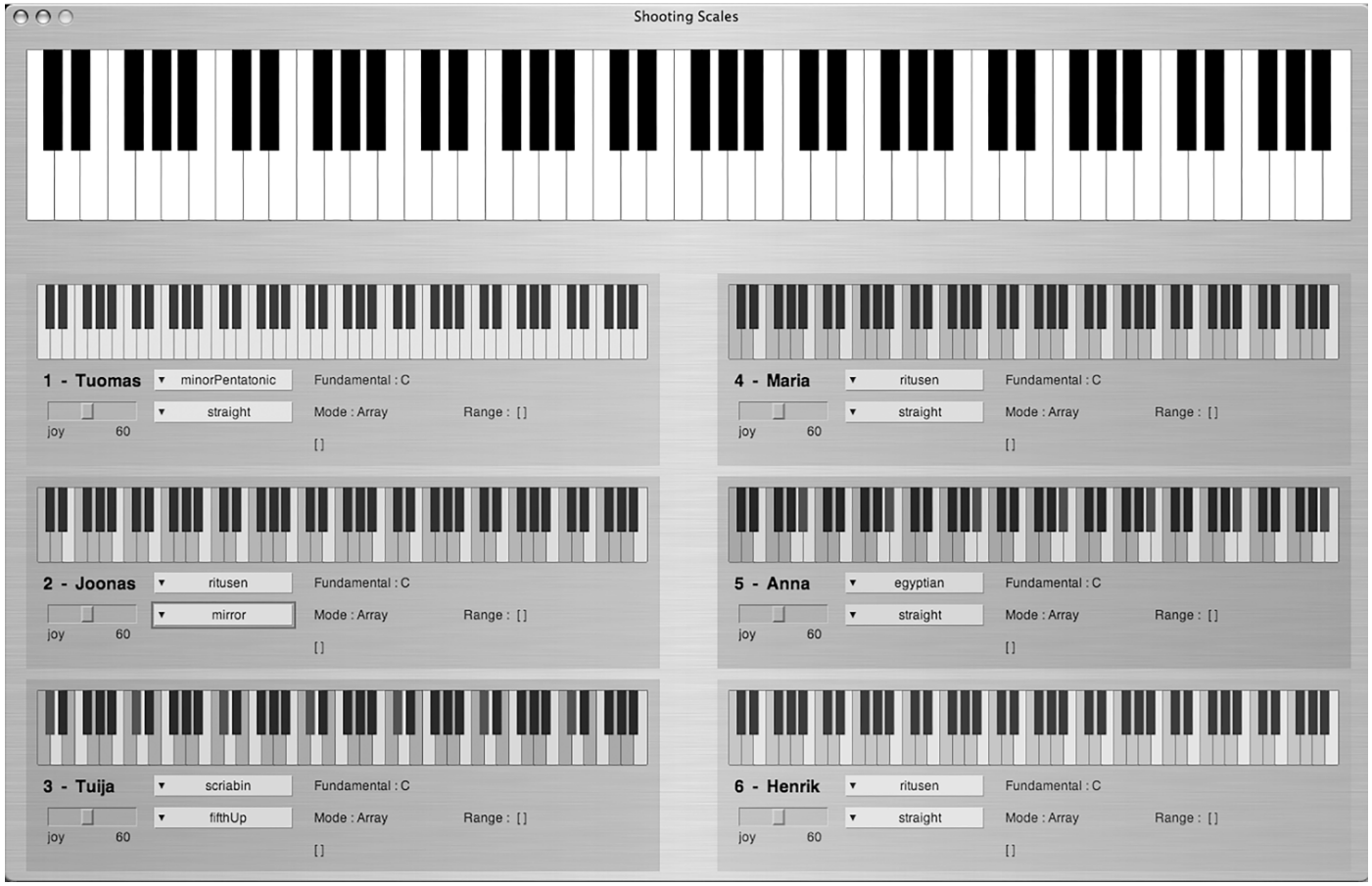


Figure 21.11
A screenshot of the Shooting Scales performance instrument.

21.4.3 Study 3: From a Sketch to an Instrument

A look out the window at snowflakes on a cold winter day may yield the desire to create a simulation of snow. In SuperCollider, we can draw snowflakes using `Pen`, but we can also check and see if the `MultiSliderView` could do it well enough. (This would probably be more convenient in terms of processing power, as the `MultiSliderView` is implemented as a primitive.) Figure 21.12 shows a simple patch that has four layers of `MultiSliderViews` with transparent backgrounds, so one can see through to the next layer of snow. When the snow lands, it triggers a bell-like sound.

```
(
var win, msl, trigAction, snowloc, speeds, speed, layers=4,   snowc
ount = 62;

SynthDef(\snowBell, {| freq=440, amp=0.4, pan=0 |
   var x, env;
   env = EnvGen.kr(Env.perc(0.001, Rand(550,650)/freq, amp),   done
Action:2);
```

```
   x = Mix.fill(6, {SinOsc.ar(freq*Rand(-10,10), 0,   Rand(0.1,0.
2))});
   x = Pan2.ar(x, pan, env);
   Out.ar(0, x);
}).add;

// fill an array with arrays (number of layers) of locations
snowloc = {{rrand(0.38,1.5)} ! snowcount} ! layers;
// fill an array with arrays (number of layers) of step size // (sp
eed)
speeds = {{rrand(0.01,0.018)} ! snowcount} ! layers;

speed = 0.1;

win = Window.new("snow", Rect(11, 311, 520, 240), border:   false).
front;
win.view.background = Color(0.14,0.17,0.24);

msl = Array.fill(layers, {|i|
        MultiSliderView.new(win, Rect(-1, -1, 522, 242))
              .strokeColor_(Color.new255(rrand(22,35),rrand(22,35)
,rrand(22,35)))
              .fillColor_(Color.new255(rrand(222,255),rrand(222,
 255),rrand(222,255)))
              .valueThumbSize_(rrand(2.8,3.8))
              .indexThumbSize_(rrand(2.8,3.8))
              .gap_(5)
   });

// when the snow falls this happens. (pitch is mapped to index // a
nd amplitude to speed)
trigAction = {arg drop, amp; Synth(\snowBell, [\freq,   400+(drop*2
0), \amp, amp, \pan, rrand(-0.8, 0.8)])};

t = Task({
   loop({
        snowloc = snowloc.collect({|array, i|
              array = array.collect({|val, j|
                   val = val-speeds[i][j];
                   if(val< 0.0, {val = 1.0; trigAction   .(j, speed
s[i][j]*10)});
                   val
```

```
                });
                array
            });
            /*
            Task uses the TempoClock by default so we need to "defer" t
he GUI updating (Function:defer uses AppClock) This means that the
 Task is essentially using the SystemClock and therefore the timing
is better on the sound front. The AppClock (used for GUI updates) h
as worse timing.
            */
            {layers.do({|i| msl[i].value_(snowloc[i])})}.defer;
            speed.wait;
    });
}).start;

// on stopping the program (Command/Ctrl + dot) the task will // st
op and the window close
CmdPeriod.add({t.stop; win.close;});
)
```

Figure 21.12
Snjókorn—a patch with four layers of `MultiSliderView` triggering sounds.

I tend to work from this type of bottom-up approach. This sketch is interesting enough to explore a bit further. After adding some functionality, it has turned into a full-blown instrument (called *sounddrops*) that is now distributed as part of the ixiQuarks. It uses the `bufferPool` tool of the ixiQuarks system, so when the drops land, they can trigger a sample, various synthesis types, code (allowing for live coding and morphing the instrument into something else in real time) or envelope an audio signal running through any audio bus. (Other ixi instruments could be outputting their signals on, say, audio bus 20 and the sounddrops would then listen to them.) As seen from this example, each slider in the array can have its own speed, and the `Task` takes care of recalculating the drop location according to another `Array` (namely, speed). We have thus taken a simple sketch and created a powerful tool that can be used for complex sequencing of polyrhythmic temporal structures that allow ease of control, pitch mapping, and, most important, a graphical representation of the process that can inspire the musician. (See figures 21.13 and 21.14.)

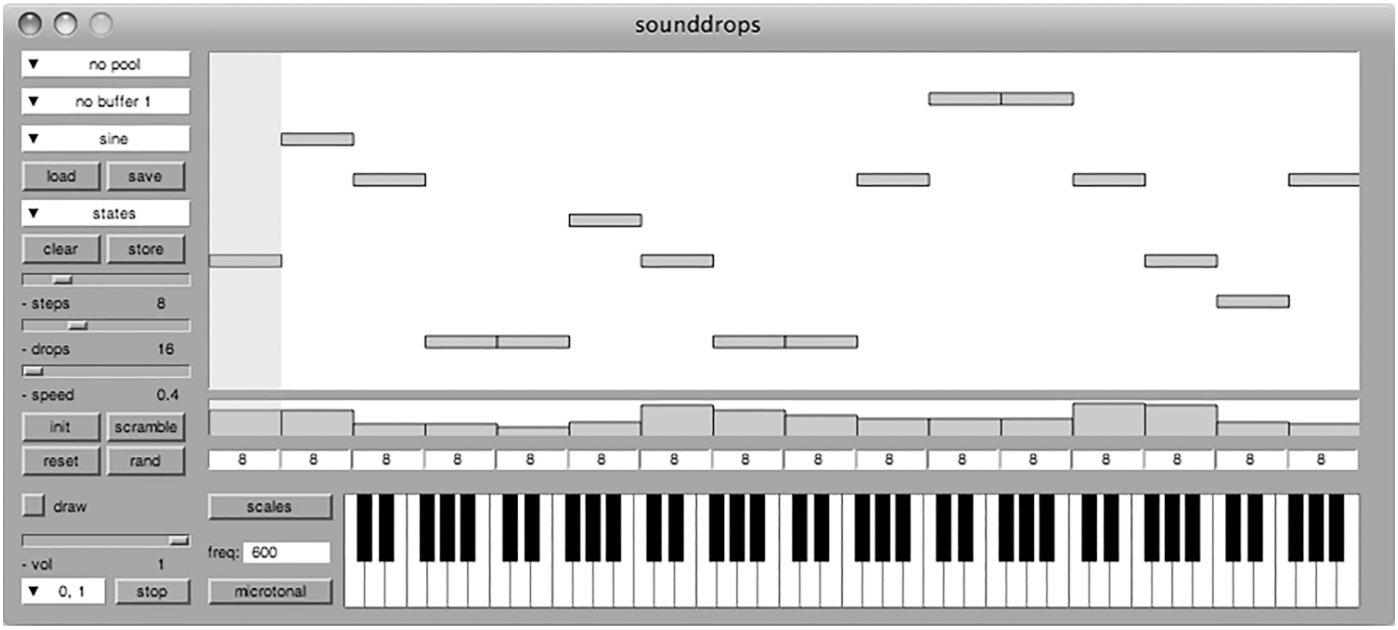


Figure 21.13
A screenshot of *sounddrops* in microtonal mode, in which each of the drops (ranging from 2 to 48 in the view) has properties such as sound function (sample, synthesis, code, or audiostream), pitch, amplitude, speed, and steps. The microtonal keyboard under the multislider view has seven octaves (vertical) and from 5 to 48 notes (equal tempered tuning) per octave.

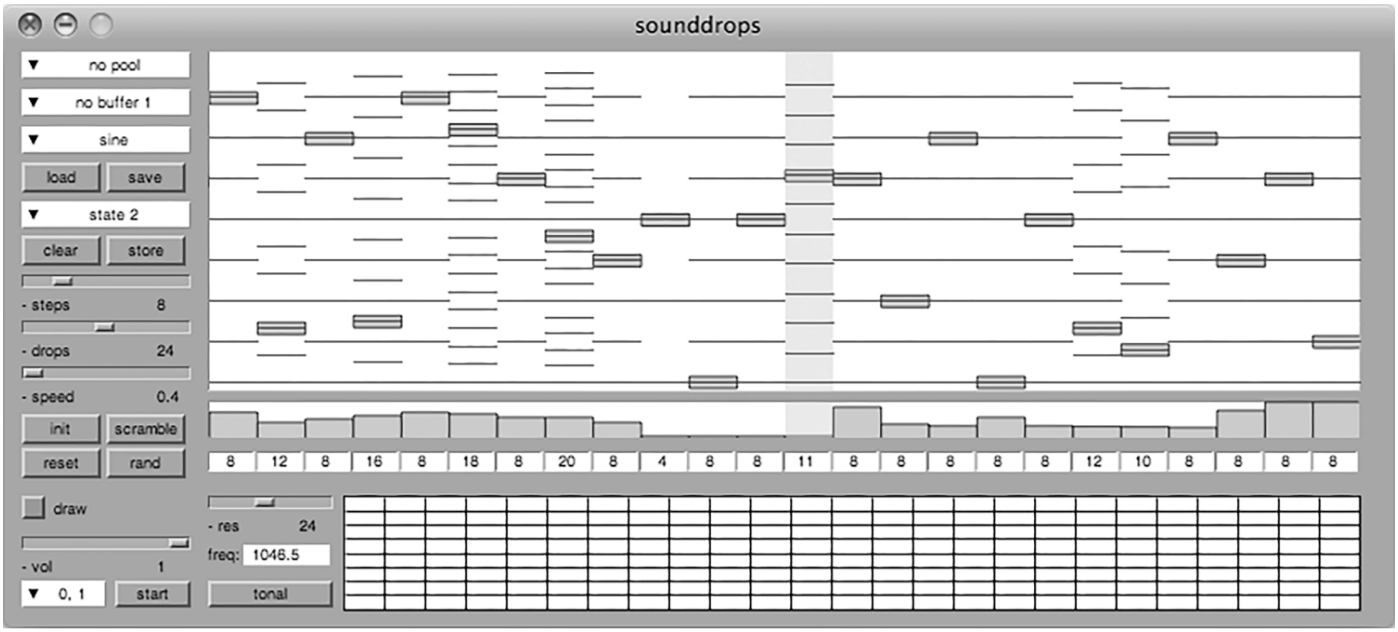


Figure 21.14
Shown here is *sounddrops* in tonal mode. There are two pop-up windows as well: one is the coding window for live coding, and the other is a window that contains various scales and chords that can be used to color the keyboard. These keys can then be used to assign the frequencies to the drops.

All SuperCollider users eventually come up with their own design for organizing buffers, buses, effects, patterns, tools, and instruments. The ixiQuarks is only *one*

approach to such organization. I decided to modularize the code so instruments could use the same buffer pools, input signals from each other, and output on audio buses that contain effects and filters. When the system is modularized in this way, the process of turning a sketch (such as the snowflakes above) into a full-blown instrument that works with ixiQuarks might not take more than a few hours.

21.5 Conclusion

In this chapter, we have seen how the environment inspires innovation through play and exploration of its affordances. It has been argued that in artistic programming, the ability to work from a bottom-up design is important, and the fact that SuperCollider is an elegant, object-oriented, interpreted language makes it extremely well suited for such experimental coding. The more protocols, hardware, and graphics the environment supports, the more it lends itself to inspirations in which the creative mind and the tool interact in a way that can be seen as improvisation, sketching, exploring, building, and composing a complete piece or tool.

The ixiQuarks are GUI instruments built on top of SuperCollider. They can be seen as creative limitations that reside on top of an ocean of potential expression. Magnusson and Hurtado-Mendieta (2007) showed that people like to have constraints and limitations of all kinds when working in creative environments. This is true not only for tools or instruments, but also for music or film theories. SuperCollider is an extremely open and expressive environment, perhaps the ideal intellectual partner for the musical freethinker. The ixiQuarks are on the other end of the spectrum: they focus, they concretize, and they constrain. They prime the user's mind into a certain way of thinking, a way that can be beneficial for the musical process or creativity. It is here that the power of the ixiQuarks lies: to allow one to be able to sketch and perform intuitively and quickly with tools that one has mastered but still be within the context and expressive scope of SuperCollider itself.

References

Holland, S., K. Wilkie, P. Mulholland, and A. Seago. 2013. *Music and Human–Computer Interaction*. London: Springer-Verlag.

Jensenius, A. R., and M. Lyons. 2017. *A NIME Reader: Fifteen Years of New Interfaces for Musical Expression.* Cham, Switzerland: Springer.

Jordà, S. 1995. "Digital Lutherie: Crafting Musical Computers for New Musics' Performance and Improvisation." PhD thesis, Department of Technology, Pompeu Fabra University, Barcelona.

Magnusson, T. 2007. "The ixiQuarks: Merging Code and GUI in One Creative Space." In *Immersed Music: Proceedings of the 2007 ICMC Conference*, Copenhagen.

Magnusson, T. 2009. "Epistemic Tools: The Phenomenology of Digital Musical Instruments." PhD thesis, University of Sussex, Brighton, UK.

Magnusson, T. 2010. “Designing Constraints: Composing and Performing with Digital Musical Systems.” *Computer Music Journal*, 34(4): 62–73.

Magnusson, T. 2019. *Sonic Writing: Technologies of Material, Symbolic and Signal Inscriptions*. New York: Bloomsbury.

Magnusson, T., and M. E. Hurtado. 2007. “The Acoustic, the Digital and the Body: A Survey on Musical Instruments.” In *Proceedings of the 2007 NIME Conference*, New York, pp. 94–99.

McCartney, J. 1996. “SuperCollider: A New Real Time Synthesis Language.” In *Proceedings of the International Computer Music Conference* (ICMC’96), pp. 257–258.

22 SuperCollider in Japan

Takeko Akamatsu

22.1 Introduction

In the foreword to this volume, James McCartney has presented the origins and history of SuperCollider (SC); I will present a specifically Japanese perspective. This chapter provides a case study of some Japanese users who have found SuperCollider a productive and empowering environment for audio art.

McCartney's aesthetic is demonstrated by SuperCollider's real-time capability, flexibility, and elegance as a programming language, which maximize the computer's potential. In turn, the combination of Japanese aesthetics with SuperCollider has made for some very interesting and exciting projects.

22.2 The History of SuperCollider in Japan

Let's trace the history of SuperCollider in Japan. According to McCartney, the first Japanese person to purchase a copy of SuperCollider 2 was Masayuki Akamatsu, a media artist. Among individual countries, Japan was the world's second-largest market for SuperCollider after the United States although other countries were close: the United States, 51.5 percent; Japan, 8 percent; United Kingdom, 7.6 percent; Germany, 6.1 percent; Canada, 3.8 percent; Australia, 3.4 percent; France, 3 percent; Italy, 3 percent; others, 13.6 percent.

Since SuperCollider 3 is now distributed free of charge, it's hard to tell how many people are currently using it in Japan, although many musicians and programmers are interested in SC, and the forum is active. After tracing the history of SuperCollider in Japan, I will relate some of the activities of today's SC users.

The first known demonstration of SC in Japan was (ironically) at Max Night, held at ICC (NTT Inter Communication Center) in Shinjuku on February 21, 1997, as the preopening event. ICC is at the vanguard of media art centers using progressive technologies in Japan. At the event, Akamatsu gave a tutorial on programming and computer music. Though Max was featured in the lecture, it had no audio synthesis capability at the time, so Akamatsu introduced SC as a real-time audio synthesis tool and demonstrated it with a live performance. His band, gaspillage, featured SC on the

CD *Maze and Lights* (1998), using it to generate abstract sounds clustered by `Spawn`, and for sound morphing. While Akamatsu was the first Japanese person to buy a copy of SC (purchased in London), the first to buy it in Japan was Yasuhiro Otani, the other member of gaspillage.

22.3 How I Discovered SC, and My Activities as an SC Maniac

After this event, many computer musicians became interested in SC; I was introduced to SC2 by a friend. I tried several sample programs and was attracted to the sounds. But I didn't even know what programming was, much less understand English, so I had to abandon learning it for the time being.

After this, I attended a progressive school of media arts, the International Academy of Media Arts and Sciences (IAMAS). I gained some experience in programming and had good opportunities to learn English. (IAMAS was the first school to buy academic licenses for SC in Japan.) Yet at that time, no introductory books (such as tutorials) or information about SuperCollider was available in Japanese. English was difficult for me, and I also had to learn the technical terms used in programming and audio synthesis. So it was very difficult for me to understand the Help files of SuperCollider. In addition, few people were using SC.

So I took a year to translate the Help files of the language and all the Help files for unit generators into Japanese. Through the process, I became fascinated with SuperCollider not only as a music production tool, but also just for itself. I published an email newsletter and created a mailing list to increase users. I crafted socks with embroidered programming code to show that SuperCollider not only was a programming language for computer nerds, but it also was fashionable (figure 22.1). I held a small weekly workshop, SuperCollider Dame School, in Nagoya (2003) and a DSP Super School with James McCartney as a guest lecturer at IAMAS in Ogaki City (2004). I've since continued with this kind of activity as an SC maniac, running the forum and wiki in Japanese (figure 22.2) and organizing SC user meetings.

Figure 22.1
SuperCollider socks.

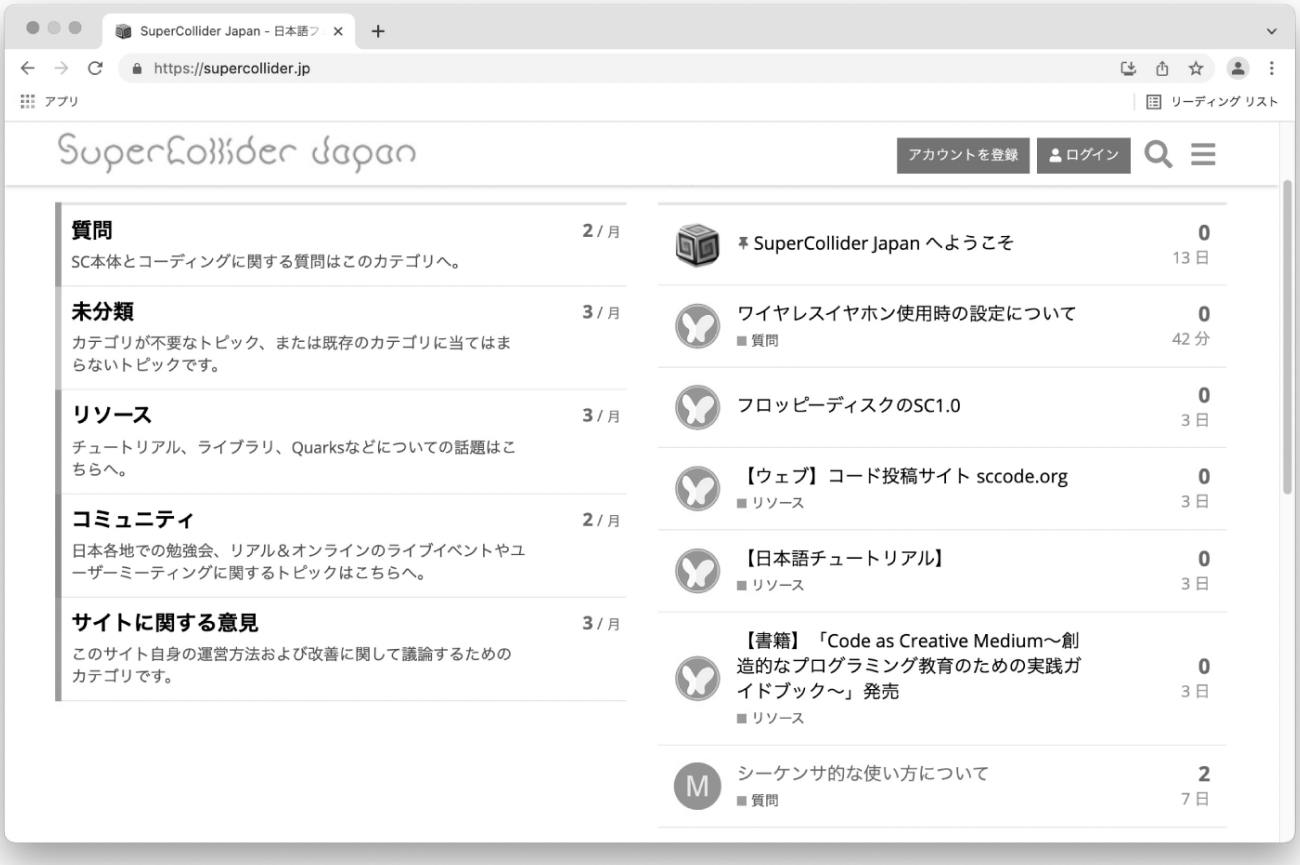


Figure 22.2
Top page of http://supercollider.jp, consisting of a wiki, forums, and more information.

22.4 SC Users in Japan

I'd like to introduce some remarkable SC users in Japan and their programming code. The code is provided in the book code repository.

SC is used as a tool for learning audio synthesis at a few schools. The most active place is Tama Art University, where Professor Akihiro Kubota teaches. In his online book *Introduction to Code Composition* (http://www.idd.tamabi.ac.jp/~kubotaa/icc/), he introduces compositional programming, algorithmic operation, and elements related to the aesthetics of digital music. He also shows how digital practice is connected to many historical methods concerning the sense and ideas of sound and music. He finds SuperCollider the most suitable tool to illustrate such ideas. Part of this material is included in the Akihiro_Kubota folder on the website.

Some of his students have been working intensively with SuperCollider. Koichiro Mori's recent open-source on-going project msplr (https://github.com/moxuse/msplr) is an application for audio sampling which works with multiple clients. It uses SuperCollider as the back end and an Electron web user interface.

Atsushi Tadokoro (https://yoppa.org/) is committed to education on sound programming and creative coding. He has been teaching at the Maebashi Institute of Technology and has published books for the learner. Besides teaching in academic fields, he has been actively performing as a notable live coder using Tidal Cycles (https://tidalcycles.org/) and SuperCollider as powerful sound sources.

SuperCollider is not used just by people with an academic background. Yamato Yoshioka is a notable SC user in Japan. During an interview, he said that he studied programming by himself when he was a child in order to build computer games. He has since been making tools for music with SC (see the Yamato_Yoshioka folder). One of his works, "Fireworks," provided the sound effects for a school play. An animation element was added later. Some elements of the sound and animation (e.g., the size and location of fireworks) are synchronized. Another work, "GoodNose," is a rock guitar simulation played from a laptop; in it, he reflects on the possibilities and controversies raised by playing music from an audio programming language.

Takuro Hishikawa, aka umbrella_process, has been making music with SC since 2004 (see the Takuro_Hishikawa folder); he also helps me moderate the SuperCollider wiki and forums. In "Rainy Route 79," he obscures the perspective of time by means of sound. With increasing saturation of sound, it is hard to know how much time has passed, like during a boring drive on the highway on a rainy day. For this piece, it is important to write chords that have no relation to each other; for this purpose, SuperCollider is convenient for him. When I quizzed him on this, he justified it by the hard work required to write many unrelated chords by hand; using SuperCollider, variations of the `rand` method are a natural way to proceed.

I myself also make synth pop using the pattern functions of SuperCollider. I've been working with the iPhone to control SC (figure 22.3). There are two important points in this work. One is that this wireless, small, and flexible device allows me to play anywhere, providing an alternative to staring at my laptop monitor when I'm doing live performance. The other is wanting to know what the essential elements and identity of my music are, which I explore by randomizing musical elements such as pitch, duration, and rhythm. SuperCollider's `Pattern` classes and functions are useful for implementing this, and it's also easy to write a program to control SC via these kinds of devices using the OSC message style. (See the Takeko_Akamatsu folder in the code repository.)

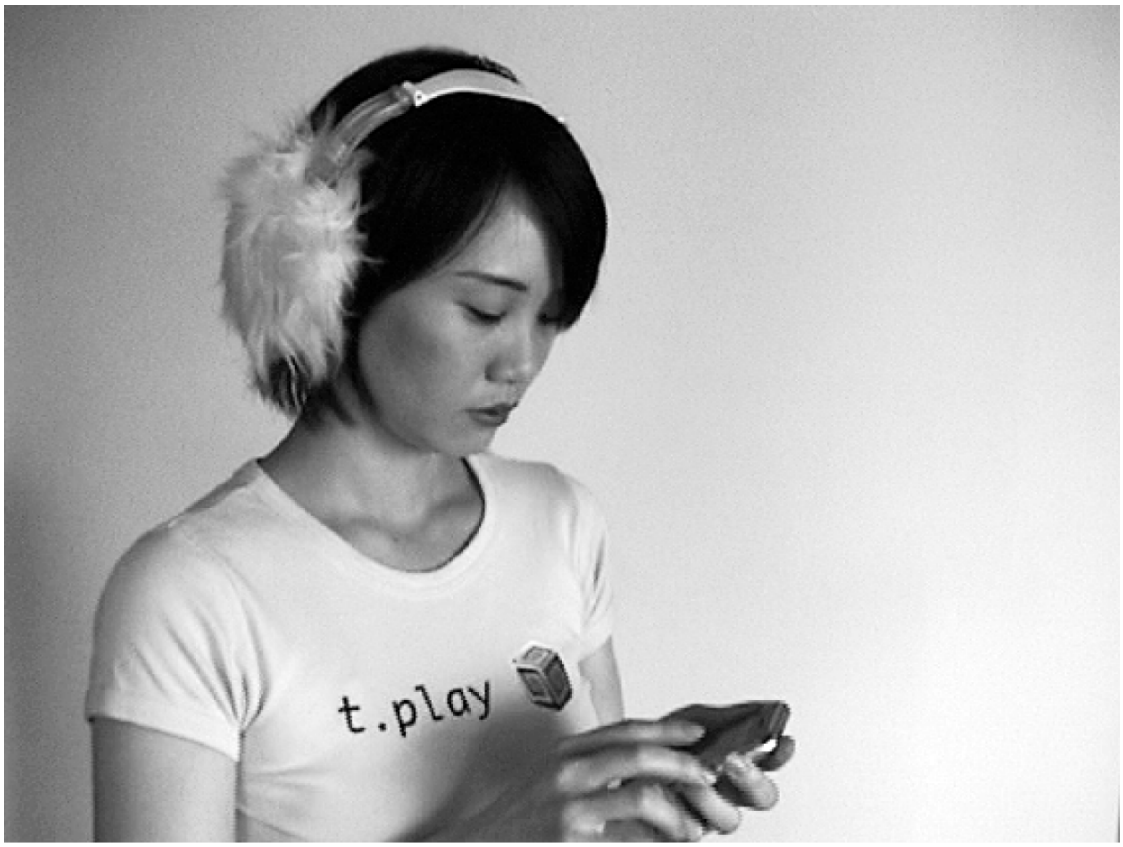


Figure 22.3
Playing SC on an iPhone.

SuperCollider has attracted an ever-increasing number of users. Compared with my first encounter with SC, it's now much easier for beginners to learn, especially here in Japan, with translated documents, tutorials, and the forum. Despite the high language barrier, we all long for the wonderful unique music that can be made using SuperCollider.

23 Dialects, Constraints, and Systems within Systems

Julian Rohrhuber, Tom Hall, and Alberto de Campo

A language is a dialect with an army and navy.
—Max Weinreich, 1945

SuperCollider (SC), like the layers of a Russian *matryoshka* doll, is a system of systems, containing an ever-increasing number of specialized subsystems that range from the general purpose to the highly idiosyncratic. Extensions to a language add both new possibilities and new constraints. This chapter will explore this idea, looking at a variety of ways in which these features and limitations can influence our practice and understanding of the process of programming and art. By discussing examples from different areas of research (e.g., language design, hardware, scheduling constraints, text manipulation, etc.), we will show how the restrictive potential of such systems can lead to an iterative and collective artistic process. Indeed, a SuperCollider composition can be regarded as a work of art within a work of art or a language within a language. We have chosen examples for their interesting idiosyncrasies, as well as to convey an impression of the dimensions that such systems unfold.

The benefits that these subsystems bring outweigh the initial difficulty of grasping their manner of operation. In programming (as in artistic activities in general), the need for constraints is at least as important as the desire for features; limitations are themselves features that require implementation. This consideration is revealed especially in the blurring of the distinction between a tool and its outcome, an application and an artwork or a model. The more general, generative, or aleatoric a work becomes, the more it becomes an environment, a system of axioms, or simply material for new works of art within its own world.

23.1 Dialects

It has often been stated that there is no essential difference between a language and a dialect; gradually one manner of speaking crosses over to another. Where languages end and dialects begin is a matter of perspective. Furthermore, languages are used heterogeneously: many people are multilingual, Creole languages are abundant, and within a single language, domain-specific vocabularies exist to facilitate communication (or, of course, restrict it) within a given situation.

A language or dialect constrains what is thinkable. To what degree this is true is a part of a long-standing debate within philosophy and linguistics (see, e.g., Wittgenstein, 1958; or Whorf, 1956). Certainly, a language's structure and vocabulary pose a limit on what we can say explicitly and how we make sense to others. Computer languages occupy a peculiar position here: on the one hand, they are an elaborate form of control instructions specifying what a machine should do, and on the other hand, they should also be human readable so as to make clear what the computer should do and how it should be achieved. A computer program is thus *doubly readable*. Donald Knuth, arguing in favor of programs as *literature*, puts it this way: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do" (Knuth, 1992, p. 99). Historically, the parallels to language were central to computer science early on (Nofre et al. 2014), but computer code itself was only rarely construed as art before the 1990s, even though computer code by both Max Mathews and Iannis Xenakis was included in a famous anthology of music notation edited by John Cage (1969). How a particular notation may affect our human interpretation of a notation or (here, computer) text is outside the scope of this chapter (for more, see Hall, 2007), but we will return to text systems in a slightly different context in section 23.5.

One may want to argue that the same program may be expressed in different ways and the choice of language is unimportant once it is in use. A program is not necessarily a *black box* (i.e., a closed system in which the user has access only to input and output). Within its development, maintenance, or composition, the specific way in which a language is structured makes some things easy to express and do, and others harder; thinking only within a given language, some ideas may never occur. Moreover, computer languages not only intertwine process and description—doing and saying—but they also enable the construction of other languages or systems, which in turn become the bases for new programs. If a program is something like a recipe for a cook, it resembles just as well a recipe for a recipe, or even a recipe for the construction of a cook.[1]

Historically, many programming languages evolved from an extension of an existing one, adding new concepts until this extension became something quite different from the initial language. (SuperCollider, for instance, started as the MAX plug-in *Pyrite*. For many other examples, see Wexelblat, 1981.) Since the 1970s, many computer languages have been designed with this in mind in order to make it easier to implement domain-specific subsystems—or even new languages with their own syntax. Code may produce code for other programs—a simple example is a *quine*, a program that returns its own source code. One example in SuperCollider is

```
(_ + '.(*' + quote(_) + '! 2)') .(* "(_ + '.(*' + quote(_)   + '!
 2)')"! 2)
```

Programming an interpreter amounts to programming a language. It is a program that executes instructions written in a specific language. For languages with a very simple syntax, such as Lisp, it is relatively straightforward to write such an interpreter in the original language itself. (See, e.g., the *metacircular evaluator* in Abelson et al., 1996, ch. 4.) The idea of object orientation takes this a step further: every object is in itself like a simple interpreter. The messages sent to such an object are actually small programs, instructions for behavior that can mean very different things depending on who receives them. This *polymorphism*—different objects responding to the same message in different ways—makes for one of the most interesting features of integrating systems within the system (see chapter 8). In SuperCollider, it is trivial to write, for instance, a kind of `List` class that behaves like any other List, except that it responds to requests to access its items with a blurring of boundaries between that item and its neighbors. Here, we create a subclass of `List` that overrides the `at` message and modifies the index that is passed in as an argument:

```
VagueList: List {
    at {|index|
        ^super.at((index + 1.rand2).clip(0, this.lastIndex))
    }
}
```

The original message at is now understood differently by the object, which returns an element at or near the index that it was asked for. For example, `a = VagueList[0, 1, 2, 3, 4]; a.at(2);` may return 1, 2, or 3. From this, it becomes clear that a subclass of an object is indeed like a very small sublanguage, or dialect, using the same syntax but with altered semantics. What is the characteristic constraint here? At first glance, we have introduced a kind of malfunction: when using this list to play a melody, some of the notes will be omitted and others repeated, without our knowing in advance which ones will be which. This intentional ignorance allows a different perspective on the musical material: while seen as *outside-time* material (Xenakis, 2001, pp. 155–161), the list of notes is identical; as *inside-time,* it displays a characteristic degree of vagueness.[2] From a purely behavioral perspective, this state consists of truly ambiguous objects.

Our discussion began with a shift from the unavoidable constraints implied in language and the variability of dialects to programming languages that encourage a formation of systems within systems. The above example of message-passing polymorphism demonstrated how we can minimally shift the semantics of a program in order to change our idea of what we are working with, for instance, a melodic pattern. By constraining access, new meaning is created.

Before discussing a number of SuperCollider subsystems, we now look at different levels at which such systems can come into play and show how apparent limitations in one area may reveal potential in another. From program text to sound, there are many intermediate levels at which interactions, but also conceptual constraints, may occur. The *code*, usually represented as a string object, is interpreted and results in a graph of connected *objects*. These objects specify a certain behavior: the *processes* of the running program that may create new objects, but also new code. In the current SuperCollider implementation, there is a distinction between very different types of processes: the nodes in the sound synthesis tree on the SC server, and the streams and schedulers on the sclang side. Beginning with the former, we discuss a brief example of a constraint on each level.

23.2 Emulators, Deficient Synths, and Appropriating Protocols

The challenges of writing efficient code and inventing faster algorithms to run on current technology have often required clever "hacks" to get the most out of any given hardware. While scientific knowledge of algorithms is advanced through such efforts, some artists deliberately return to old hardware for other reasons. Historically, a game music specialist could often tell what computer was used for a piece, not only from its sound but also from the compositional strategy employed to work within the limited means available.

The aesthetic dimensions of such hardware constraints are well illustrated in a simulation of ENIAC, the first electronic computer, by Martin Carlé. The SuperCollider implementation uses the quasi-continuous synthesis model of UGen graphs to implement ENIAC's analog circuits. While one could encapsulate a universal machine entirely in a single UGen, the ENIAC model consists of a network of UGens simulating the hardware as the components change states in the process of running a program. In the example below, the ENIAC Cycling Unit (which was used for synchronizing the other units) is modeled as a `CU_PulseLookUpTables` UGen, which expects as input a periodic ramp from 0 to 80 (volts, as it were) to cycle through its "addition time" cycle and creates 10 synchronized clock signals, which in turn control the timing of all the other ENIAC components (see figure 23.1). These are also modeled as UGens and include multipliers, dividers, square-rooters, and accumulators (with 10 decimal places, thus running at the equivalent of 33.22-bit precision). All the UGens were first modeled in MATLAB/Simulink and from there exported for automatic conversion to SuperCollider UGens. The following demonstrates the use of the emulator:

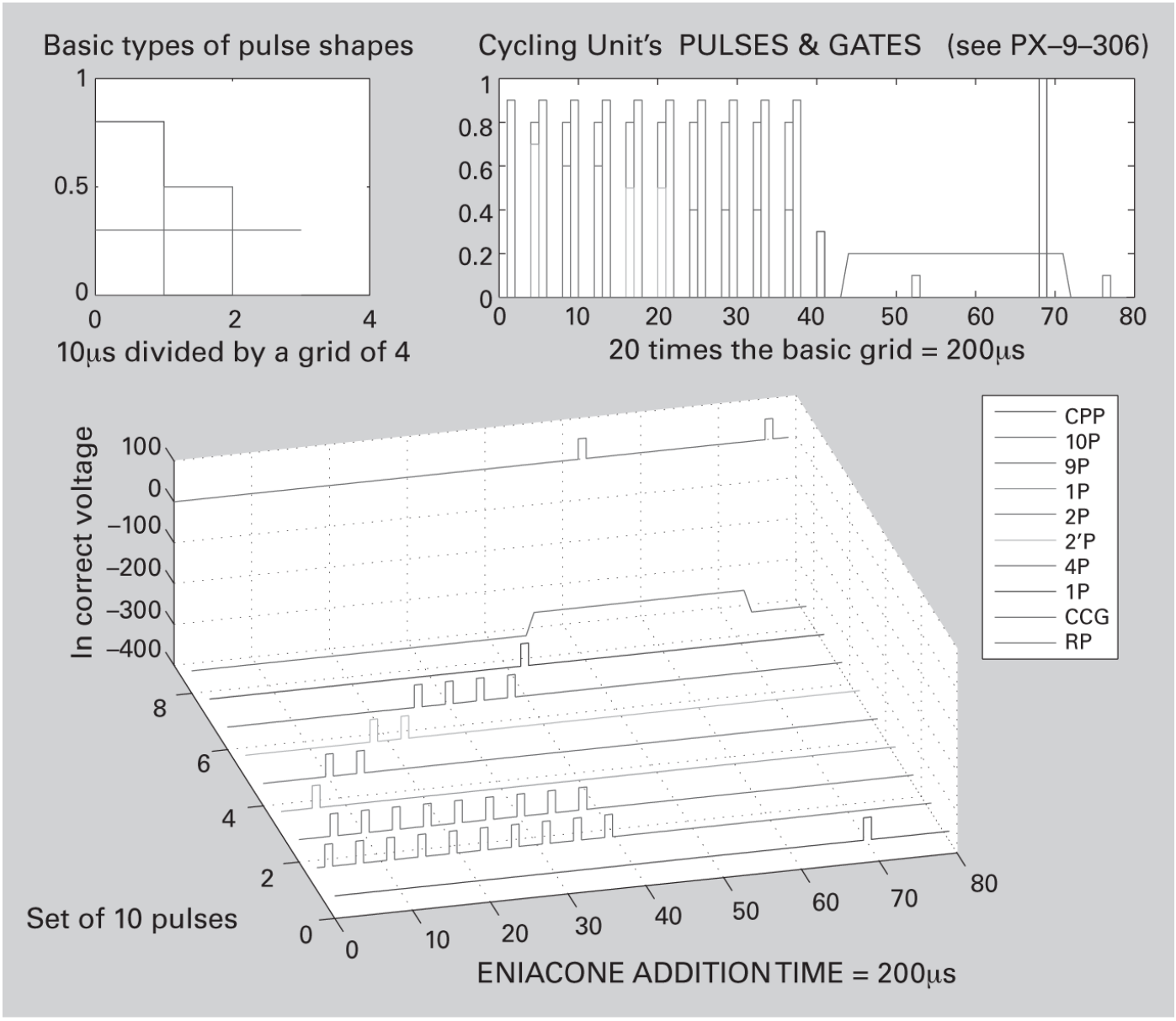


Figure 23.1
The ENIAC Cycling Unit and the structure of the 10 pulses that it creates.

```
// Eniac Cycling Unit with adjustable clock speed
(
{   var clockspeed = MouseX.kr(1, 300, 1);
   var clocksignal = LFSaw.ar(clockspeed).range(0, 80);
   var timingPulses = CU_PulseLookUpTables.ar(clocksignal);   // 10
channels.
              timingPulses * 0.2
}.scope;
)
```

Hardware constraints have their equivalents in software. A program or a synth definition that has a peculiar parameter mapping, a specific trade-off, or some life of its own can be regarded as a constraint, or a system within the system. Within the graph of a synth, for instance, it is possible to derive boundary conditions or limits for some parameters from others or to control several properties with a single one. To give a very basic example, when frequency and pulse width are coupled, timbre depends on frequency:

```
{
   var freq = MouseX.kr(20, 2000, 1);
   Pulse.ar(freq, freq.explin(20, 2000, 0.95, 0.05))
}.play;
```

This dependency can become more intricate if a different mapping function is used:

```
{
   var freq = MouseX.kr(20, 2000, 1);
   Pulse.ar(freq, freq.explin(20, 2000, 0, 5pi).sin * 0.45 +   0.5)
}.play;
```

In such a way, instead of trying to make as many parameters accessible for control as possible, we can find ways to reduce the number of entry points. On the other hand, we could supply both width and frequency inputs but still make them depend on one another, for example by a limit that they pose for each other or by a mutual influence of their values when changed (figure 23.2).

With these simple prototypes in mind, we consider how compositions are made of parts that set basic constraints or rules for other parts. While in some respects, a score for flute is written with the idiosyncrasies of the instrument in mind, a programming language like SuperCollider renders these relations more complex. What used to be called *control* can be situated anywhere between rule definition (constraint) and navigation (exploration of consequences). In such a way, and often initially unnoticed, a constraint turns into a subsystem. One such almost invisible subsystem in SuperCollider becomes apparent in the composition of synth definitions: the UGen graph is constructed by the operations within sclang, but its processes run on the synthesis server, which cannot access objects such as functions or dictionaries.

Another genre of constraint is the simulation of one computer by another, as discussed previously in relation to ENIAC. Many other older chip sets also have been implemented in the virtual form of emulators in order to run old programs written with a very different set of features in mind. Examining such a system in SuperCollider demonstrates well how a part of a larger system can work with its own peculiar rules.

`Pokey`, a UGen written by Fredrik Olofsson, is an oscillator that implements an essential set of features of the 8-bit Pokey sound chip that formed part of the ATARI computer and served as sound source for many 1980s computer and arcade games.[3] While the floating-point inputs of the `Pokey` UGen can be modulated at the control rate, they are truncated to integers, which are then interpreted as binary numbers. Not only does the resulting sound conjure up memories of "jump-and-run" games and the subsequent 8-bit or "chiptune" music genre, but its implementation has another kind of cultural resonance; `Pokey` employs a very specific set of functions that bear a resemblance to the intersections of numerical sieves used by Xenakis in a number of his compositions (Xenakis, 2001, pp. 268–288; written in 1971). The sometimes surprisingly complex relation between addition and multiplication is used to generate a continuum between cyclic and pseudorandom movements—to this end, both Xenakis and Pokey superimpose several layers of numerical regularities, resulting in interference patterns of common multiples.

```
(
{
    var f = {| a, b | [a.min(1-b), b.min(1-a)]};
    var freq = f.value(MouseX.kr, MouseY.kr) * 400 + 500;
    SinOsc.ar(freq) * 0.1
}.play;
)
(
a = {|freq=100, width=0.5|
    var df = freq-LastValue.kr(freq);
    var dw = width-LastValue.kr(width);
    freq = freq + (dw * 100);
    width = width + (df / 100);
    Pulse.ar(freq, width.clip(0.01, 0.99).poll) * 0.1
}.play;
)

a.set(\freq, exprand(200.0, 600.0));
a.set(\width, 1.0.rand);
```

Figure 23.2
Two ways of constraining parameters.

A first glance reveals the abundant limitations within `Pokey`'s functionality. There are only four sound channels for polyphony; each channel has 8-bit frequency resolution (with 256 steps), representing integer divisions of the sample rate, and 4-bit amplitude

(with 16 steps). Also, each output channel is simply an amplitude-modulated pulse wave (figure 23.3). In its arcade game version, `Pokey` has been used for mimicking physical sound events, such as airplanes, Geiger counters, fires, cars, crashing buildings, and waterfalls, simply by supplying two bytes, one for frequency and one for sound. This type of constraint may resemble a bank of presets or `SynthDef`s, which present us with no more than an arbitrary collection of choices. Yet this oscillator is different mainly for two reasons. First, the sounds have a very specific relation to each other, as they are all produced by the same method: a high-frequency pulse train is subdivided (every *n*th trigger may pass, controlled by the frequency byte) and then is fed through a combination of numerical sieves that essentially consist of a pseudorandom pattern of holes. Because some of these polynomial pseudorandom patterns repeat fairly early, regular interfering beating patterns result—a small change in the frequency byte results in a sound that is almost entirely unlike the previous one (see figure 23.4).

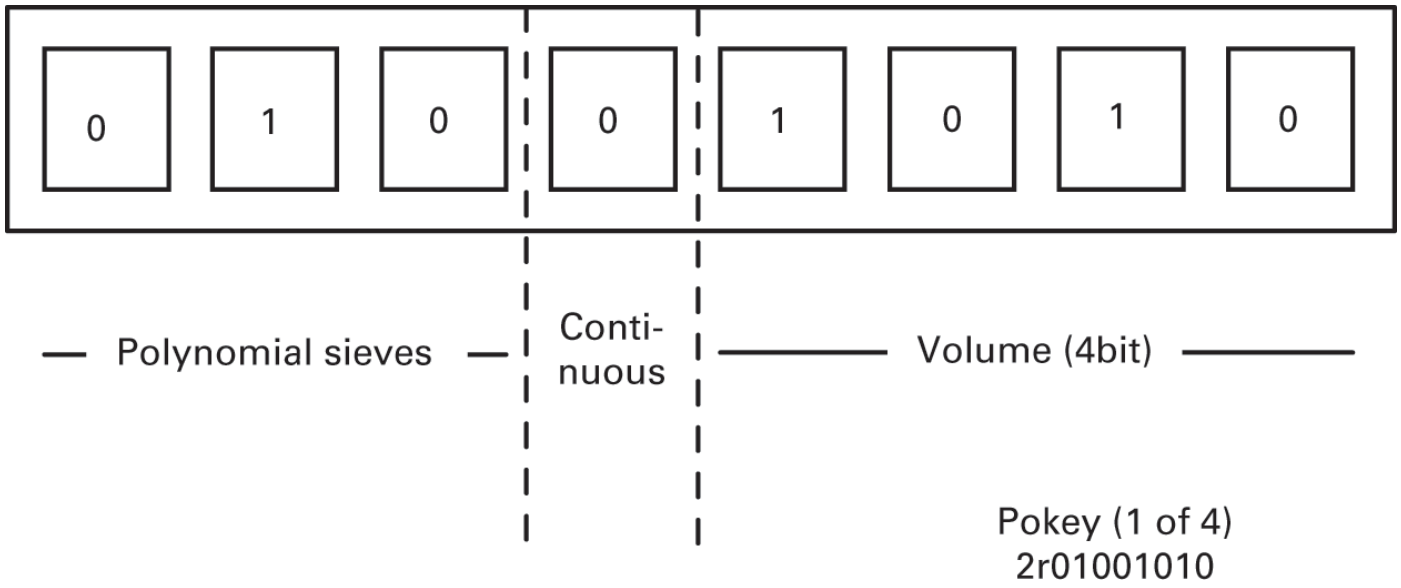


Figure 23.3
Overview of the registers for one `Pokey` channel.

```
// modulating the frequency input to a Pokey UGen results in great
 variance
(
{
   var rate = MouseX.kr(0, 255);
   var mod = LFPulse.kr(1);
   var amp = 2r1100; // 12 of 16
   Pokey.ar(rate + mod, audc1: 2r01000000 + amp);
```

```
}.play
);

// modulating the pure tone bit
(
{
   var rate = MouseX.kr(0, 255);
   var mod = LFPulse.kr(1);
   var amp = 2r1100; // 12 of 16
   Pokey.ar(rate, audc1: 2r00100000 + (mod * 2r00100000) + amp);
}.play
)
```

Figure 23.4
Modulating Pokey inputs.

As shown above, `Pokey` differs from a set of presets in the way parameters closely interfere with each other to form a kind of sonically constrained inner logic. The second reason for `Pokey`'s difference is illustrated in figure 23.3. This graphic shows how, by using a binary numerical representation (e.g., 2r01001111 instead of the equivalent 79), some of this inner logic affects the way that it is coded: by varying a certain bit in a number over time, for instance, we can multiply a binary number by the modulating unit generator. The term "addition" carries a different meaning here, and the mapping becomes nonlinear: adding up to 4 bits increases the amplitude, but adding more than that will affect the sound in a different way. Someone who knows the intricacies of this chip can interleave its specific aesthetics with the rest of the system.

`Pokey` is an example of a metaphor of a specific chip within SuperCollider which demonstrates two things: internally, a system within a system can be an interconnected space to explore compositionally, and also, externally, it may give rise to other ways of connecting it to other parts of the system.

Insofar as programming systems provide means to go beyond an expected purpose, use and misuse are not always distinguishable. Reflecting on the computing environment *Lively*, Daniel Ingalls et al. (2016) wrote, "It was tempting to consider repurposing this language and the web browser to recreate the conditions for creative programming in the context of what was becoming a universal platform." It has been possible to run the SuperCollider synthesizer in a web browser since 2020.[4] Repurposing the web browser as a programming system is not simply a convenience, it is a system within a system that brings about new constraints and possibilities. Below we briefly outline some experiments by Rohan Drape employing different web browser–based SuperCollider dialects.[5]

JsSc3 is a library for communicating with the SuperCollider synthesizer using the JavaScript interpreter of the browser (Wirfs-Brock, 2020). It lets one define signal processing graphs and send them, along with any related instructions, to the synthesizer to be played. *Spl* is a simple programming language that also runs in the browser and resembles but isn't the same as the SuperCollider language. The *Spl* program `{SinOsc(IRand(48, 72).MidiCps, 0) * Rand(0.05, 0.1)}! 2` has the expected meaning.

SuperScript and *Small Hours* are the editors for the *JsSc3* and *Spl* programs, respectively. They're loosely modeled on the SuperCollider 2 system, which encourages literate programming using ordinary word processing conventions and implicit document linking.

SuperCalc is a spreadsheet interface for writing dynamic graphs of communicating *Spl* programs. In order to replace parts of a signal graph while it is running, the graph must be partitioned and the individual parts named. Spreadsheets are interesting in this context because they automatically assign each program part a separate text editor, along with a spatially coherent name by which the parts may refer to each other. Using the spreadsheet paradigm also brings a programming practice from other areas of work, such as accounting, into experimental sound synthesis.

Block SuperCollider is a visual editor for *Spl* programs. Block editors are a family of visual programming systems that use interlocking graphical blocks to represent the elements of a program.[6] There are blocks for variable assignment and reference, procedure definition and application, and logical and mathematical operators. In this system, there are also blocks for the standard components of a synthesizer. Figure 23.5 depicts a translation of one of the examples from the SuperCollider 2 Help files.[7]

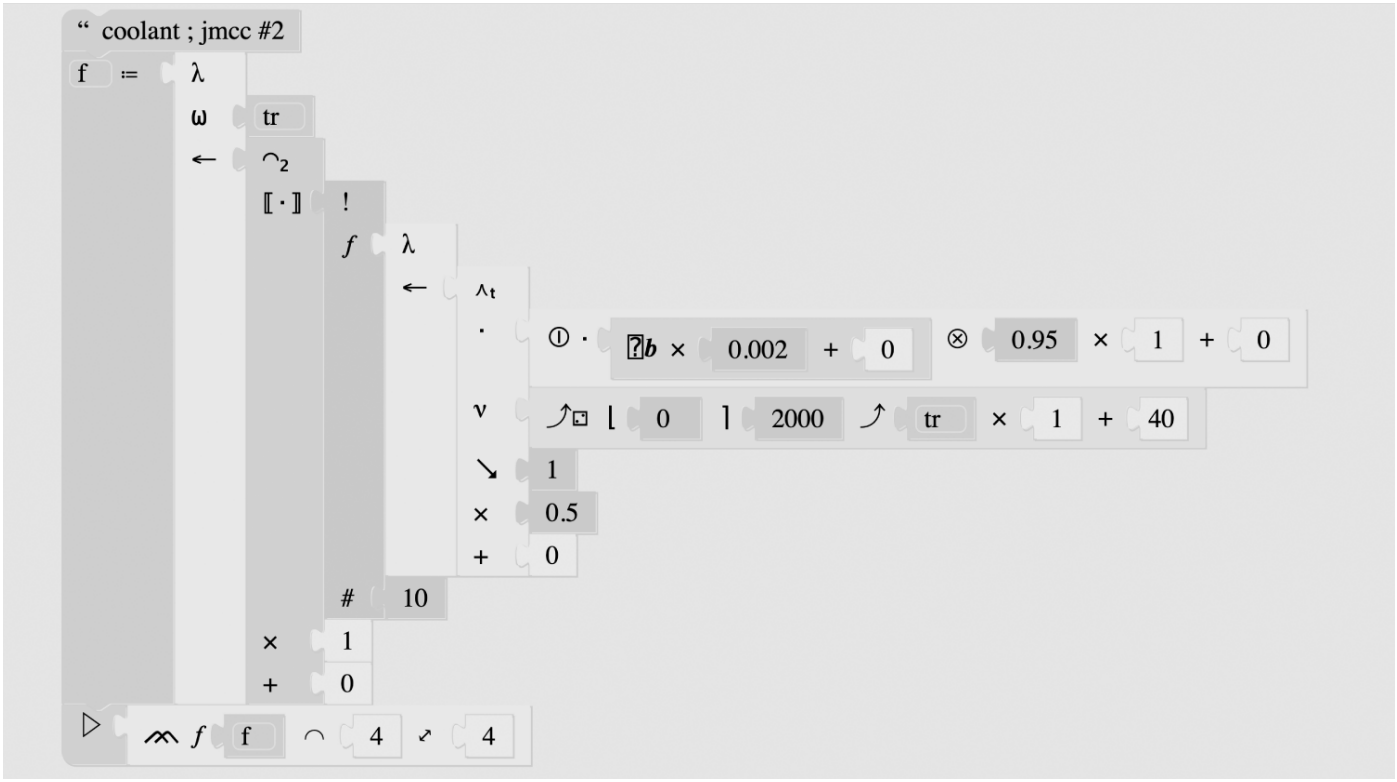


Figure 23.5
Block SuperCollider “coolant” example.

When a block drawing is evaluated, it prints itself as an *Spl* program, which is then evaluated in turn. The names displayed in the block drawings can be different to the names printed in the program text. In this drawing, the Spl names are given symbolic forms, making drawings concise while also naming each input parameter. (Blocks have help texts that can be viewed by hovering the mouse above the block, and these show the printed names and the names of the input parameters.) Some names in this drawing are ◠2=*Splay2*, ʌt=*Ringz*, ·=*input*, ν=*frequency*, ↘=*decayTime*, ⓘ=*OnePole*, ?b=*BrownNoise*, ⤴⚀=*TRand*, ⤴=*trigger*, and ⩙=*OverlapTexture*.

The colors indicate the category or kind of the block. In this drawing, there are distinct colors for oscillators, filters, random number generators, panners, procedures, variables, numerical constants, and comments.

Block systems are interesting because they are two-dimensional drawings of programs, a form of “very rich text editing.” They open up many possibilities for combining in one place both the definitions of sounds and their real-time performance controls. They can also help make complex systems, such as SuperCollider, approachable and comprehensible. The browser as “universal platform” offers many opportunities for experimenting with nontraditional systems for both authoring SuperCollider programs and distributing completed works.

23.3 Scheduling Constraints: HierSch

Textbooks often claim that programming is about modeling aspects of the real world. As an approach to synthesis algorithms, physical models introduce material constraints into sound synthesis. By formalizing their parameters, they allow both exploration of their state space and their combination with other entities (Cook, 2002). What we hear are the boundary conditions of human ideas about the physical environment.

Such models are plentiful among the default SuperCollider UGens, and more are available for download from the sc3 plug-ins project. *Sonification*, for instance, is a wide field at the borderline between reflection on models and exploration of the world (discussed further in chapter 13).

The next example shows a method for prioritizing simultaneous events. It models a common physical constraint of musical performers such as percussionists—namely, that they have a limited number of simultaneous means with which to strike instruments.

A key aspect of any compositional system concerns the relations and constraints among its musical elements (Babbitt, [1961] 2003). Most systems, including tonal systems, are governed by a number of structural or perceptual hierarchies. Temporal relations between musical elements also form an important part of the constraints of the traditional tonal system in Western music; rules exist as to which simultaneities (chords) may follow which. Chords are in turn articulated within a metrical hierarchy, which in much Western music connotes strong and weak beats in the bar (Lerdahl and Jackendoff, 1983; Temperley, 2001). Where compositional systems are used for improvised or semi-improvised music, it can be a challenge to form a coherent system of (implicit or explicit) rules governing the articulation of the system's elements, especially where those elements (be they performed live or computer generated) have a degree of independence.

The `HierSch` class (the name comes from *Hier*archical *Sch*eduler) by Tom Hall is designed to explore a real-time SuperCollider approach to such issues in the temporal domain. (`HierSch` is available as a Quark.) A minimalist rule-based framework for priority-based scheduling, `HierSch` uses the same scheduling interface as `TempoClock`, but with an important difference: in `HierSch`, streams of functions, the events of which may or may not sound, can be independently scheduled according to their relation to other simultaneous events in other scheduled streams. This relation is expressed in the form of a number of simple rules centered on a numerical priority assigned to each stream (or individual tasks within a stream).

Priorities in `HierSch` range from 1 (highest) to 12 (lowest); priority 0 is used to mark layers that always play. Once scheduled, by default an item with a given priority will be evaluated only if another item with a higher priority is not also scheduled to play at that moment. Results can be varied by inverting the priorities, increasing the number of

priorities that can play simultaneously, and so on. `HierSch` has a number of scheduling methods, including schedAbs, whose syntax is similar to its namesake in `TempoClock`. One of the main syntactical differences is the addition of a number of extra arguments in `HierSch`'s `schedAbs`, including priority and stream, which supply the control information for determining, together with `HierSch`'s internal state, which events will play and which will be ignored.

Figure 23.6 illustrates a simple example of concurrent `HierSch` schedules, in which simultaneous functions are considered by the system for evaluation in three places, but only the highest (shown darkened in the figure) are evaluated.

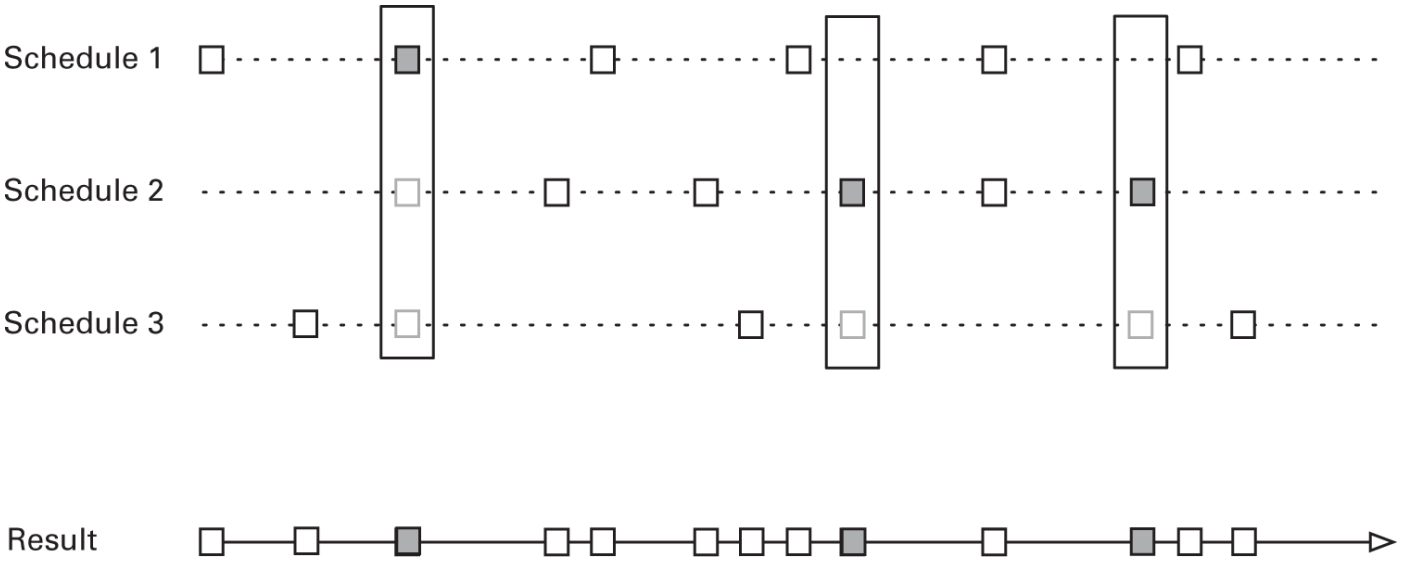


Figure 23.6
`HierSch` scheduling constraints and priority levels.

A simplistic diatonic example of the kind of scheduling constraints illustrated in figure 23.6 is demonstrated in figure 23.7. The example also shows the use of the counterargument within the scheduling method. This is passed into the function and used to select the appropriate `Array` element pitch offset from the priority 1 and 2 functions. The `VagueList` class discussed in section 23.1 is used in the lowest voice for limited temporal pitch displacement.

```
(
SynthDef(\ping, {
   arg out = 0, mfreq = 69, pan = 0, gain = 0.2, dur = 0.25;
   Out.ar(out, Pan2.ar(
   SinOsc.ar(mfreq.midicps, 0,
   EnvGen.kr(envelope: (Env.perc(0.01, dur)), doneAction: 2)),
   pan, gain));
```

```
}).send(s);
// function to play a synth
m = {|f, d=0.3, g=0.2, p=0| Synth(\ping, [\mfreq, f + 45, \pan, p,
\gain, g, \dur, d])};

// function to make a chord
c = {|a, b, c| [a, b, c].do{|i| m.value(i, 1.2, 0.075,   rrand(-1.
0, 1.0))}};

t = TempoClock.default.tempo_(116 / 60); // assign clock to t
b = HierSch.new(t); // start new HierSch, pass in clock
)

(
// HierSch schedules
b.schedAbs(t.beats.ceil + 48, 0, {var offset = [0, 5, 7, 12];   c.v
alue(*[12, 16, 19]+ offset.choose)}, Prand(#[1.5, 3], 30));   // en
ters last, priority highest

b.schedAbs(t.beats.ceil + 14, 1, {|b, p, d, c| m.value([0, 0,   7,
 5, 4].at(c % 5) + [12, 24].choose, 0.4, 0.15, rrand(-1.0,   1.
0))}, Pseq(#[2, 2, 2, 1], 15)); // enters middle, priority   middle
b.schedAbs(t.beats.ceil, 2, {|b, p, d, c| m.value(VagueList[0,   1
2, 4, 7, 10, 10, 9, 9, 7].at(c % 9))}, Pseq(#[0.5, 0.5,   0.5, 0.5,
0.25, 0.75, 1, 0.5, 0.5], 17)); // enters first,   priority lowest
)
```

Figure 23.7
Priority-based `HierSch` scheduling.

23.4 Object Systems: Redirections and Constraints

Since this book provides many examples of class libraries and varieties of object semantics, we will take only a short excursion to two specific aspects of this topic. First, we will show how to use a *redirection* of access and assignment to create little worlds with rules of their own; second, we will show an example of a *special syntax* for declaratively formulating solutions to problems.

23.4.1 Redirecting Assignment: Maybe and LazyEnvir

As discussed in connection with the `VagueList` example, message passing is a way to change behavior polymorphically. By reinterpreting a message differently, we are able to integrate new functionality smoothly to propagate through the system. Two very basic

things are excluded from this principle, though: a variable is not an object, and assignment is not a method. In other words, in the expression `x + 1`, the variable `x` is not itself the receiver of the message +, but rather its *value* (whatever it is). Likewise, in the expression `x = 1`, the = is not a message but a sign for the binding of a value to `x`. This provides the foundation of the object system: behavior is bound to variables by objects. Environments have a similar syntax; we can also write `~x = 1`. In actual fact, however, we are implying something else: `~x = 1` is really equivalent to writing either `\x.envirPut(1)` or `currentEnvironment.put(\x, 1)`. Within the just-in-time programming library (JITLib), discussed in chapter 7, there is a simple class called `LazyEnvir`. Its superclass is `EnvironmentRedirect`. (`ProxySpace` also derives from it.) Instead of simply setting the value named *x* in the environment, a lazy environment returns a proxy, a placeholder instance of the class `Maybe`. In a loose analogy to the programming language Haskell, in SuperCollider, a `Maybe` allows us to write *underspecified* calculations within a lazy environment (see figure 23.8). More generally, this concept is called "lazy evaluation."

```
p = LazyEnvir.push;
~a = ~b * ~c;
~a.value; // => nil
~b = Pseq([1, 2, 3]).asStream;
~c = 10;
~a.value; // => 10
~a.value; // => 20
~b = [1, 2, 3];
~a.value; // => [10, 20, 30];
~a.postcs;   // => Maybe((Maybe([1, 2, 3]) * Maybe(10)))
p.pop
```

Figure 23.8
`Maybe` yes.

Any object that responds to `source_(arg)` by keeping the argument in some internal representation and returning it when sent the message source, can be used in a `LazyEnvir` to modify how assignment and reference work. The proxy classes discussed in chapter 7 are examples of such objects.

23.4.2 Declaring Constraints: List Comprehensions

The SuperCollider language can be cryptic, but unusual syntax often is simply a sign of a programming language influence from far outside the mainstream. Expressions such as `_ + 1! 7` (which is the same as writing `Array.fill(7, {|i| i + 1})`) have their own

history and are yet another example that SuperCollider is already a system of systems drawing from many sources (in this case, APL and Haskell). Some programming languages provide a way to pose constraints for solving problems, whereby after performing some algebraic and logic reformulations, the system returns solutions. One way of formulating such problems are *list comprehensions,* in which we supply a range of possible values for each variable (a domain) and a relation between these variables. For instance, to request all coprimes (all integer pairs that have only 1 as a common divisor) between 2 and 10 in SuperCollider, we write

```
f = {:[x, y], x <- (2..10), y <- (x..10), gcd(x, y) == 1},
```

where `x <-(2..10)` stands for all numbers between 2 and 10 and `y <-(x..10)` represents all numbers between each x and 10. To constrain this domain, we declare that the greatest common divisor of `x` and `y` must be 1. Here, `f` can produce all the solutions step by step: `f.next` returns `[2, 3]` and so on. More terms can be added:

```
f = {:[x, y], x<-(2..10), y<-(x + 1..10), gcd(x, y) == 1 and: x.isP
rime. not and: y.isPrime.not}
```

In figure 23.9, we add an additional constraint such that the numbers should not be ordinary primes.

```
(
var x;
x = {|rates=#[1, 1]| Ringz.ar(Impulse.ar(rates) * 0.1, rates * 80,
1 / rates)} .play;
fork {
   var str = {:[x, y],
        x<-(40..2),
        y<-(x + 1..40),
        gcd(x, y) == 1,
        x.isPrime.not and: y.isPrime.not
   };
   0.5.wait;
   str.do {|primes|
        x.setn(\rates, primes.postln);
        (primes.product / primes.sum / 20).wait;
   }
};
)
```

Figure 23.9
Coprimes as frequency and trigger rates.

Now the first solution is `[4, 9]`. Such sequences of solutions have many applications—and being essentially streams, they can be used in the SuperCollider pattern system or just called within a routine (see figure 23.9).

There are many musical subsystems that use constraints of a similar kind, albeit with different kinds of notations, some of which can be partly combined. To mention only a few, the *crucial library* provides constraint objects that could, for example, be used to select musical styles from databases. An event-based constraint system is used to experiment with Javanese music rules in the ethnomusicological research project Virtual Gamelan Graz (Schütz, Rohrhuber, and de Campo). Nick Collins's Infno synth pop generator uses dynamic constraints for interlocking multiple instrumental layers. In Andrea Valle's GeoGraphy project, sequences of sound objects are formalized and visualized as graphs of path constraints, somewhat like a petri net.[8] Graphical interface constraints form the basis of Thor Magnusson's projects (discussed in chapter 21).

23.5 Text Systems

Thus far, we have discussed different levels at which to implement systems within systems: *primitive*, *processes*, and *object* semantics. A further possibility, less common in SuperCollider (perhaps for good reason), is the *textual* level itself. Given that computers are ideal tools with which to manipulate and analyze text, it is not surprising to see early computer analysis and recent recompositions of Samuel Beckett's partly algorithmic text *Lessness*,[9] as well as John Cage's use of the computer to aid the creative "writing through" of a diverse range of existing texts (Coetzee, 1973; Drew and Haar, 2002; Cage, 1990).

Before we show how to include other computer languages at the level of the text, we examine how strings can be used to express alphabetical constructions a little differently while remaining within the realm of standard code. A string such as `"aggaca"` or `"what else?"` may be given very different meanings. For instance, `a = Pseq("aggaca".ascii)` describes a stream that returns the number series 97, 103, 103, 97, 99, 97 (`a.asStream.all`); among other things, such a stream may be realized as a melody or a specification for a resonator bank. Using dictionaries (or in more advanced cases, finite state machines), strings may be split into parts with a meaning (e.g., to evaluate functions). Figure 23.10 shows how to do such a translation with a varying key size.

```
(
var dict, maxLength = 0;
```

```
dict = (
   ab: {(note: [4, 0, 7], legato: 0.1, dur: 1)},
   ba: {(note: [4, 9, 8], legato: 0.3, dur: 0.3)},
   aaa: {(note: 5, legato:1.5)},
   bbb: {(note: 0, legato:2.5, dur: 0.25)}
);

dict.keys.do {|key| maxLength = max(maxLength, key.asString.size)};

f = {|str|
   var i = 0, n = 0, substr, event;
   while {i < str.size} {
        substr = str[i..i + n];
        event = dict[substr.asSymbol].value;
        if(event.notNil) {
             substr.postln;
             i = i + n + 1;
             n = 0;
             event.postln.play;
             event.delta.wait;
        } {
             if(n + 1 < maxLength) {n = n + 1} {n = n-1;   i = i +
1}
        };
   };
};
)

// play some sequences
fork {f.value("abbbbaab");}
fork {f.value("aaabbbabbaaaabbabaaaaba");};
```

Figure 23.10
A very simple notation translator.

There are many possibilities for alternative notations, file readers, and string modification methods. Regular expression methods such as `findRegexp` and `matchRegexp` may even be considered little languages in themselves. (See also chapter 26 for the use of strings and ASCII codes to denote symbolic musical notation.) Implementations of *symbolic machines* such as simple finite state machines (e.g., the `Pfsm` pattern), instruction synthesis (`Instruction`, in Nick Collins's SLUGens), or Emil Post's *TagSystems* (e.g., `Dtag` UGen by Julian Rohrhuber) show that a simple

syntax system may cause surprising and intractable behavior. For general-purpose programming this is an unwanted constraint (a *Turing tar pit*), but the situation can be otherwise in experimental mathematics or sound synthesis.

Because everything is an `Object` in sclang, its own interpreted code is represented as a `String` object that also may be modified directly. In order to use a string of code in a program, we can define it (`x = "1 + 2"`) and interpret it explicitly: `x.interpret/3;` returns the result `1`. This also means, for example, that we may replace the plus sign with a minus sign (`x = x.replace("+", "-");`) so the second part would now return `–1/3`. We can easily replace the plus sign with nonsense code so that the results will be difficult to understand or invalid—an obvious disadvantage of being able to self-modify code. Nevertheless, such techniques are used wherever the notation syntax itself needs to be modified, for instance, in code art pieces, in which the form of representation is part of the meaning; for experiments with language syntax; and for the integration of other languages into SuperCollider code, as discussed below.

The integration of other languages with SuperCollider code is enabled through the existence of a *preprocessor* that allows the Interpreter to modify code before interpreting it (see chapter 20). This kind of code for modifying other code may simply be a function that takes the compile string as the argument and returns the modified code. Continuing from our earlier example, `this.preProcessor = {|str| str.replace("+", "-")}` changes every plus sign to a minus sign in SuperCollider code. (Since this doesn't make much sense from a programming point of view, breaking as it does much existing code, it is useful to know how to return to standard behavior: `this.preProcessor = nil`.) Extending this principle on a less trivial level, the Quark named `PreProcessor` is designed to help embed multiple languages into SuperCollider (`"PreProcessor".include;` to activate it, evaluate `this.preProcessor = PreProcessor.new`). Using `PreProcessor`, the code between the delimiters `<%` and `%>` is not evaluated directly; instead, it returns a function that takes an environment in which both language and input values may be defined (see the `PreProcessor` Help file). Once evaluated, it stores the result in this environment. An interpreter for the Turing-complete esoteric programming language *brainfuck*[10] has been implemented (see an example on the book website), but `PreProcessor` is also expected to be useful as an interface with scientific systems such as Octave (octave.org). In line with our earlier discussion of dialects, the form, and especially the *syntax,* of notation are not neutral to the direction that our thinking may follow. Problem-specific little languages (Bentley, 1986) often can be conveniently implemented within a host language simply by adding objects and methods, with no need to modify the host language syntax. Some little languages add very little new functionality, but they do modify the general behavior of the host language. As an example, the little language `GLOBOL` (see the Quarks directory) has a different syntax "look and feel" compared to sclang, and it introduces specific

constraints and features. Mainly, these are the constraints that global variables are always audio proxies and the feature that these audio proxies run on all networked machines. Thus, `A = {SINOSC:AR(B) * 0.2}` becomes a named global 8-channel audio proxy which can be made audible with `A:PLAY`. The global variable B can be set later with, for instance, `B = 234`, which will propagate to all networked computers running `GLOBOL`.

23.6 Systems within Systems

Programmers tend to reinvent the wheel—perhaps simply because it takes longer to find a code that fits the purpose than to write new code. But if we consider what the nature of a "wheel" is in this case, we realize that synthesis algorithms, notation systems, useful classes, and idiosyncratic objects are less like engineering tasks than works of literature, formal science, and conceptual and performative art. Alan Newell, in his discussion of fair use and patentability of algorithms, doubts that we can draw a clear line between the discovery of a law of nature, a piece of literature (which is covered by copyright), and a novel technical idea (which is covered by patent law). One of his suggestions is to "consider a model, in which inventions produce, not consumables, but 'inventibles'. That is, [to] suppose the primary effect of every product is to enable additional inventions" (Newell, 1986, Rohrhuber 2013). Without a doubt, the core of SuperCollider is a large conglomerate of such "inventibles"—whether fine-grained unit generators, design ideas such as multichannel expansion, or the Pattern library. From the perspective of art, every part and subsystem—even down to a simple method definition—may turn out to be as much a piece of art as a source of new art (McCartney, 2003); it is *meta-art* in the best sense. SuperCollider fuses not only sound synthesis with high-level programming paradigms, but also technical refinement with scientific and artistic invention, as if to remind us that these are just different aspects of culture.

Within the constraints of the GPL, every program published may at any time become the source of a new style—it may be quoted, appropriated, and modified. Barthes's notion of the "death of the author" comes to mind here (Barthes, 1977 [1967]), for at the very least, identity of invention can be seen as a continuum. Literal examples of this are the occasional *cadavres exquis* that developed on the sc-users mailing list, an example of which, *CadavreExquisNo2.scd,* can be found on the book website.

In an interview first published in 1968, John Cage discusses the 10 months of computer programming that it took him to make the piece *HPSCHD* with Lejaren Hiller. Comparing subroutines to the traditional use of chords by composers, Cage states (Kostelanetz, 2003, p. 82):

> The notion that the chord belongs to one person and not to another tends to disappear, so that a routine, once constructed, is like an accomplishment on the part of society, rather than on the part of a single individual. And it can be slightly varied, just as chords can be altered, to produce quite other results than were originally intended. The logic

of a routine, once understood, generates other ideas than the one which is embodied in it. This will lead, more and more, to multiplication of music for everybody's use rather than for the private use of one person.

Cage here highlights the important *social* aspect of computer programming implied in all this discussion, as explicitly cultivated later within the open-source movement, including the SuperCollider community. Glenn Gould (1984 [1964], p. 93) offers a related perspective on the notion of multiple authorship and "implicit multilevel participation" in electronic culture. The term "community of practice" has been developed to explore how learning and sharing of knowledge happen within such communities (Lave and Wenger, 1991). A community of practice can be described as encompassing a *domain* of knowledge, a *community* that has an interest in this domain, and a *practice* that is developed in order to be effective in it (Wenger et al., 2002, p. 27). From this perspective, SuperCollider as software can be seen as a kind of system within a system *of learning* within the SC community itself, as well as being its primary focal point of practice (Wenger, 1998). At different levels, as discussed in this book, SuperCollider code—from little code components that carefully add a few concepts to large libraries that represent a whole way of thinking—forms a continuum of means and ends. Code written with a specific purpose in mind may inform the design of a musical composition, just as a piece of music may suggest a new programming concept.

Notes

1. Such programs may be either programs that output program text or programs that can read program text and interpret it. (Examples are on the book website; see Quines and others.)
2. For a deeper exploration of vagueness, see Williamson (1994).
3. In the US patent application, the chip was described as "Apparatus for producing a plurality of audio sound effects." The UGen is based on a classic emulator by Ron Fries (1996–1998), which has been used in many applications. The `Pokey` UGen is available from Olofsson's website at https://fredrikolofsson.com/code/sc/ (accessed January 23, 2023).
4. See initial announcement at https://scsynth.org/t/webassembly-support/3037 (accessed 2022-12-06).
5. Available at https://rohandrape.net/?t=www-rd&e=e/code.md and https://github.com/rd—
6. Block SuperCollider was developed using the Blockly library https://developers.google.com/blockly (accessed January 23, 2023).
7. SuperCollider 2, for Mac OS 9, is available on the original SuperCollider website at http://www.audiosynth.com (accessed January 23, 2023). The example is in the examples 2 file as "coolant." The equivalent SuperCollider 3 code for this example is available in the book chapter's code listing.
8. Available from https://composerprogrammer.com/infno.html (accessed 2023-02-01).
9. Find this online at http://www.random.org/lessness (accessed 2022-12-06). The simple algorithm to generate a further *Lessness* has also been implemented by these authors in sclang.
10. https://en.wikipedia.org/wiki/Brainfuck (accessed 2022-12-06).

References

Abelson, H., G. J. Sussman, and J. Sussman. 1996. *Structure and Interpretation of Computer Programs,* 2nd ed. Cambridge, MA: MIT Press.

Babbitt, M. 2003 [orig. 1961]. "Set Structure as a Compositional Determinant." In S. Peles, S. Dembski, A. Mead, and J. N. Straus, eds., *The Collected Essays of Milton Babbitt.* Princeton, NJ: Princeton University Press.

Barthes, R. 1977 [orig. 1967]. “Death of the Author.” In Roland Barthes, *Image-Music-Text*. London: Fontana.

Bentley, J. 1986. “Little Languages.” *Communications of the ACM*, 29(8): 711–721.

Browne, R. 1981. “Tonal Implications of the Diatonic Set.” *In Theory Only*, 5(1–2): 3–21.

Cage, J., ed. 1969. *Notations*. New York: Something Else Press.

Cage, J. 1990. *I–VI*. Cambridge, MA: Harvard University Press.

Coetzee, J. M. 1973. “Samuel Becket’s *Lessness*: An Exercise in Decomposition.” *Computers and the Humanities*, 7(4): 195–198.

Cook, P. R. 2002. *Real Sound Synthesis for Interactive Applications.* Natick, MA: A. K. Peters.

Drew, E., and M. Haar. 2002. “*Lessness*: Randomness, Consciousness and Meaning.” Available online at http://www.scss.tcd.ie/publications/tech-reports/reports.03/TCD-CS-2003-07.pdf.

Gould, G. 1984 [orig. 1964]. “Strauss and the Electronic Future.” In T. Page, ed., *The Glenn Gould Reader.* New York: Knopf.

Hall, T. 2007. “Notational Image, Transformation and the Grid in the Late Music of Morton Feldman.” *Current Issues in Music*, 1(1): 7–24.

Ingalls, D., T. Felgentreff, R. Hirschfeld, R. et al. 2016. “A World of Active Objects for Work and Play.” In *Proceedings of the Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 238–249.

Knuth, D. E. 1992. *Literate Programming.* CSLI Lecture Notes, no. 27. Stanford, CA: Center for the Study of Language and Information.

Kostelanetz, R. 2003. *Conversing with Cage,* 2nd ed. New York: Routledge.

Lave, J., and E. Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation*. Cambridge: Cambridge University Press.

Lerdahl, F., and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge, MA: MIT Press.

McCartney, J. 2003. “A Few Quick Notes on Opportunities and Pitfalls of the Application of Computers in Art and Music.” In *Proceedings of the Ars Electronica* “*CODE*” *Symposium,* Linz, Austria, pp. 262–264.

Newell, A. 1986. “Response: The Models Are Broken, The Models Are Broken!” *University of Pittsburgh Law Review,* 47: 1023–1031.

Nofre, D., M. Priestley, and G. Alberts. 2014. “When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960.” *Technology and Culture*, 55: 40–75.

Rohrhuber, J. 2013. “Intractable Mobiles. Patents and Algorithms between Discovery and Invention.” In T. Thielmann, E. Schüttpelz, and P. Gendolla, eds., *Akteur-Medien-Theorie* (pp. 265–305). Bielefeld: Transcript.

Temperley, D. 2001. *The Cognition of Basic Musical Structures.* Cambridge, MA: MIT Press.

Wenger, E. 1998. “Communities of Practice: Learning as a Social System.” *The Systems Thinker*, 9(5). Available online at https://thesystemsthinker.com/communities-of-practice-learning-as-a-social-system/.

Wenger, E., R. A. McDermott, and W. Snyder. 2002. *Cultivating Communities of Practice: A Guide to Managing Knowledge*. Boston: Harvard Business School Press.

Wexelblat, R., ed. 1981. *History of Programming Languages*. New York: Academic Press.

Whorf, B. 1956. *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf,* Edited by J. B. Carroll. Cambridge, MA: MIT Press.

Williamson, T. 1994. *Vagueness*. London: Routledge.

Wirfs-Brock, A., and B. Eich. 2020. “JavaScript: The First 20 Years.” *Proceedings of the ACM Programming. Languages*, 4(June): 1–189.

Wittgenstein, L. 1958. *Philosophical Investigations,* 2nd ed. Oxford, UK: Basil Blackwell.

Xenakis, I. 2001. *Formalized Music,* 2nd ed. Hillsdale, NY: Pendragon Press.

24 Artists' Statements

Juan Gabriel Alzate Romero, Helene Hedsund, Patrick Hartono, Norah Lorway, Andrea Valle, Marianne Teixidó, Neil Cosgrove, Shelly Knotts, Juan Sebastián Lach, Mileece, Sam Pluta, Ann Warde, and Anna Xambó Sedó

Juan Gabriel Alzate Romero

In a world obsessed with musical linearity, adopting SuperCollider (SC) freed my mind and put me on a radical path where ideas about music theory, structure, and performance were challenged to their core. From live coding improvisations, to sctweets in 140 characters, up to full-scale performances with acoustic instruments, sensors, and video projections, SuperCollider has always provided me with the right tools and algorithms to bring my musical thoughts into the physical world.

As an improviser, starting from a blank SuperCollider document was always exciting. Boilerplate code or snippets aren't always necessary to explore ideas and convert them into sound. Starting with basic sound components, then building the musical experience from impulses and sine waves was second nature for me. Adding filters and delays and further processing them during live performance would start exposing my sonic ideas to the audience and bring me closer to the new nature of music that SuperCollider could help me discover. Multiplying these elements with random parameters and, in a matter of seconds, adding modulations and structure continued to develop any emerging ideas. A particular favorite: Filtered impulse generators at different frequencies created with just a few characters thanks to the ergonomic syntax of SuperCollider,

```
{GVerb.ar(Splay.ar(Decay2.ar(Impulse.ar({(1..4).choose}!8))))}
  .play,
```

is my way to start drafting rhythmic ideas until I can shape them further into patterns.

NodeProxies and the ProxySpace have been at the core of my musical concepts and my live coding style. My approach to exploring and achieving musical ideas has been heavily inspired and shaped by this environment. In a matter of minutes, the sound can be evolved from nothing to everything. Noise, melodies, and harmonies coexist and

interact within the nodes and can be mixed, assigned, and controlled with a few commands, or even with a generic UI for the most common parameters. Anything can be executed, reexecuted, or removed whenever the music deems necessary. There is a beginning and an end, but in between, the process of combining, transforming, adding, and removing synths, patterns, effects, and interactions in real time is something I couldn't have achieved with any other software.

In my experience with laptop ensembles and hybrid groups that include acoustic instruments, the possibilities for rehearsing, experimenting, and composing are countless, but what I have grown to enjoy the most are the unintended "mistakes" I have made. As I elaborate on my musical ideas, keeping an open mind and ear is paramount; any changes and mistakes can become part of the composition. Infinity aside, not all the music I have made is completely free from regular musical constraints; applying tempo synchronization, harmony, and key changes over the network throughout an ensemble can be a small obstacle to overcome, but once these problems are solved, all these musical constraints and structural setups can be relegated to a framework or library for added functionality, as we have done with Benoît and the Mandelbrots (https://github.com/cappelnord/BenoitLib). This leaves the way free for further creative thoughts, exploration, and improvisation within a language and environment that inspires us.

Helene Hedsund

I compose mainly electroacoustic multichannel pieces, often working at EMS (www.elektronmusikstudion.se) in Stockholm, Sweden.

I use SuperCollider in order to get away from using DAWs and the limitations they present, and instead only use my own software tools. This is possibly different from the ways many people use SuperCollider. Many of these tools are dedicated software instruments used to create one of my pieces. Somebody else could use the instrument and create their own version, but—if my built-in restrictions work as intended—a version not too different from my idea of the piece.

I find it an interesting idea that instead of asking a composer to send sound files for a concert, one could ask the composer to send a software instrument to a performer and then, after some practice, they could use the instrument to create their own performance of the piece.

I have also made generic instruments that I have reused in composing many different pieces. I've been using SuperCollider for many years now, and I still have lots of things to learn. As soon as you begin to master some parts of it, you begin to feel that the possibilities are limitless. You just have to get an idea and figure out how to do it.

Another advantage of using SuperCollider rather than a DAW is how easy it is to adapt a piece for different speaker setups. By altering some numbers, the same piece

can be played in 4, 8, 12, or any number of channels. I also prefer the simple arrangement of source material when working in SuperCollider. A directory of sound files and a few text files. That's it. Compared with working a DAW, this feels very neat and tidy. And last but not least … the fact that SuperCollider is open source frees me from the feeling of being trapped and manipulated that I often get while working with proprietary software.

Patrick Hartono

"I dream of instruments obedient to my thought and which with their contribution of a whole new world of unsuspected sounds, will lend themselves to the exigencies of my inner rhythm" (Varèse and Wen-Chung 1966, p. 11).

For the past 15 years, my musical practice has revolved around the use of SuperCollider and similar platforms. I first encountered the software in 2008, prior to starting my studies at the Institute of Sonology. During this period, experimental electronic music was largely dismissed by the music community in Indonesia, driving me to pursue my studies in Europe.

Growing up in a country with a rich musical heritage shaped by colonialism, I was motivated to find a tool that would allow me to break away from the Western classical music that I was taught as a child. I found that SuperCollider offered a blank slate, a vast potential for creative expression unencumbered by a specific musical context. In my work, I use SuperCollider as my primary tool for improvisation, incorporating elements of traditional Indonesian music, particularly the concept of Balungan (skeleton melody) in Gamelan music, into my algorithmic system.

My musical system with SuperCollider consists of four main synthesis processes. The first two serve as a continuous sound source, characterized by specific behaviors such as chaos, serving as the skeleton or rearrangement of musical patterns. The last two processes are interactive (performer-driven), generating sound based on my real-time commands, whether through live coding or controller methods. At times, these last two processes can be seen as the contrasting element, lacking any timbral connection to the continuous sound. Think of the continuous sound as a school of sardines swimming harmoniously until disrupted by a predator's approach.

While SuperCollider is algorithmic in nature, I do not use it as a generative tool to make music but as a foundation (system) to facilitate my intuitive sonic imagination, in a similar approach to composers like Ferneyhough and Sidharta.

Through this approach, I can improvise freely and create intuitive sonic textures, which I comprehend as similar to the interaction between Gamelan players. With SuperCollider, I am not limited by time-based rules or systems, allowing me to explore sound fully in a holistic dimension. The reflections that occur during Gamelan

performances at the front of Masjid Agung during the Sekaten festival in Yogyakarta have greatly influenced me and inspired me to use SuperCollider's multichannel capabilities to achieve the idea of spatialization, freeing sound to move around the space.

Recently, experimental electronic music has rapidly gained in popularity in Indonesia, with live coding being one of the strong areas of interest. Although few use SuperCollider specifically, the use of SC has raised some concerns for me. Rather than using this platform as a means of pushing forward their musical expressions, many are more focused on the programming aspect, captivated by the ability to create music through code. Nevertheless, I see this as a starting point and hope that in the future, young composers and musicians from Indonesia will balance the use of modern technology with the philosophical musical traditions of their culture, and not get exploited by the technology.

May the noise be with us!

Melbourne, Summer 2023

Varèse, E., and C. Wen-Chung. 1966. "The Liberation of Sound." *Perspectives of New Music*, 5(1): 11–19.

Norah Lorway

My work incorporates new and existing technology, often investigating how these can enable greater liveness, collaboration and accessibility in both my performance and compositional practice. I am particularly interested in bringing about awareness of surveillance, artificial intelligence (AI) ethics, and other similar tech issues which affect humans globally.

My research and creation process is predominantly focused on live coding and software development for audio; I am a live coder, algoraver, and composer of ambient and electroacoustic music, all of which features SuperCollider as either a key compositional or live performance tool. I also develop a variety of software, using SuperCollider and C++, such as Autopia, an artificial intelligence bot which acts as a musical collaborator for live coding performances (Lorway et al. 2021). Other work has been focused on networked live coding laptop ensemble performances, such as my work with the Birmingham Ensemble for Electracoustic Research (BEER) since 2011 and more recently with my music tech group, Beesting Labs.

Recent work is largely focused on exploring the grieving process in a way that allows for audience interaction and collaboration. This is carried out through programming language development for audio, video art, live audiovisual performance, and interactive virtual installations. Along with this work, I have been building Another world, a long-form audiovisual live coding performance using SuperCollider (for

audio) and p5.js. It is currently a solo work, but I intend to expand it as a collaborative, networked performance.

By building accessible software tools for music, I hope to share them to a wide range of skill levels and musical interests; SuperCollider forms an important part of this work.

Lorway, N., E. Powley, and A. Wilson. 2021. "Autopia: An AI Collaborator for Live Networked Computer Music Performance." *Proceedings of the Conference on AI Music Creativity* (MuME + CSMC), July 18–22, 2021 (Online/University of Music and Performing Arts of Graz, Austria), https://aimc2021.iem.at/wp-content/uploads/2021/06/AIMC_2021_Lorway_Powley_Wilson.pdf

Andrea Valle

https://andreavalle.org/

I was introduced to SuperCollider in 2003 by my friend Hairi Vogel. He showed me how through SC I could blend seamlessly audio capabilities and music symbolic manipulation, and SuperCollider sounded great. So I got hooked, but before diving into SC audio processing, I was fascinated by the language itself: compact, elegant, powerful. Though after some years I started using SC to manage all my music productions, consistent with my appreciation of the language, my first project with SC did not involve audio at all, as it concerned algorithmic composition outputing music notation. In the piece *Lamine d'Antigone* for six voices (2007), I used SC to integrate speech data from Praat to then generate LilyPond code; I then assembled the rendered music notation in ConTeXt, a TeX-based system. In the *Selva petrosa* cycle (2019–), I used my SC module SonaGraph to extract spectral data, then I generated MIDI files and finally imported them into MuseScore (Valle 2019). Recently, in *Miriade* (2021), I took an analogous but different path by generating audio signals in SC and mapping them into standard music notation via the music21 library in Python. In short, SC has proved to be extraordinarily fit to work not only in its specific domain of application, but also as a "gluing" language weaving together other systems. As a further example, in *Dispacci dal fronte interno* for undetermined number of strings, printers, and live electronics (2012), there is an audio feedback system listening to the sonic environment, then generating—from the environment itself—8-channel audio, but also generating music notation encoded as PostScript textual files that are sent from SC directly to printers on the stage (via shell piping), and picked up and played by the musicians. The latter's sound was then added to the soundscape that was analyzed by SC, and so on.

Thus, "gluing" means not only the capability to connect software but also hardware, resulting in a sort of heterogeneous computational network. Since 2008, I started working with physical computing. While mostly relying on Arduino as an interface toward the analog (electronic) world, I consistently used SC as the brain of all my systems, simply because I wanted an elegant and fast way to manage the high-level

control of low-level devices. So SC was the backbone of my installations *Machina logotelica, Organo fonatorio,* and *Trilobiti* (2012–2013), in which various scavenged objects interacted with the audience via old telephones, intercoms, or reversed loudspeakers. SC was used both to analyze the incoming signals from the audience and to generate and trigger the appropriate behavior in the machines by sending commands via serial port or ethernet to Arduinos.

A "summa" of these approaches is probably the four-part cycle I coauthored with Mauro Lanza, *Systema naturae* (2012–2017), for 10 musicians and electromechanical devices (Valle and Lanza 2017). Here, all my share of the symbolic composition, involving cellular automata, text sonification, genetic algorithms, and more, has been thought out and implemented through SC. But also the actual real-time control system for live playing (synchronizing human players and machines via click tracks), featuring GUIs, score representation and physical computing, is entirely managed by my faithful, reliable SuperCollider.

Valle, A. 2019. "SonaGraph. A Cartoonified Spectral Model for Music Composition." *16th Sound and Music Computing Conference (SMC2019)*, Málaga, Spain. https://doi.org/10.5281/zenodo.3249425

Valle, A., and M. Lanza. 2017. "Systema Naturae: Shared Practices between Physical Computing and Algorithmic Composition." *14th Sound and Music Computing Conference (SMC2017)*, Espoo, Finland. https://doi.org/10.5281/zenodo.1401973.

Marianne Teixidó

https://marianneteixido.github.io/

Through my work as an artistic process, I observe and interact with technology as a territory of resistance. Sound has been a constant that allows me to give vent to sensations, thoughts, and emotions resulting from reflections of the Latin American cultural and sociopolitical context, to denounce and make visible gender violence through art.

ProxySpace, as an environment, is for me a metaphor to weave and spin sounds, concepts, interfaces, and languages. I write sounds that knot and embroider textures collectively with the computer as a nonhuman agency. Live coding as a performative practice allows me to connect and disconnect these flows of sound movement that contain units of meaning that are articulated as language at the level of written text. They are resonances of sound ideas.

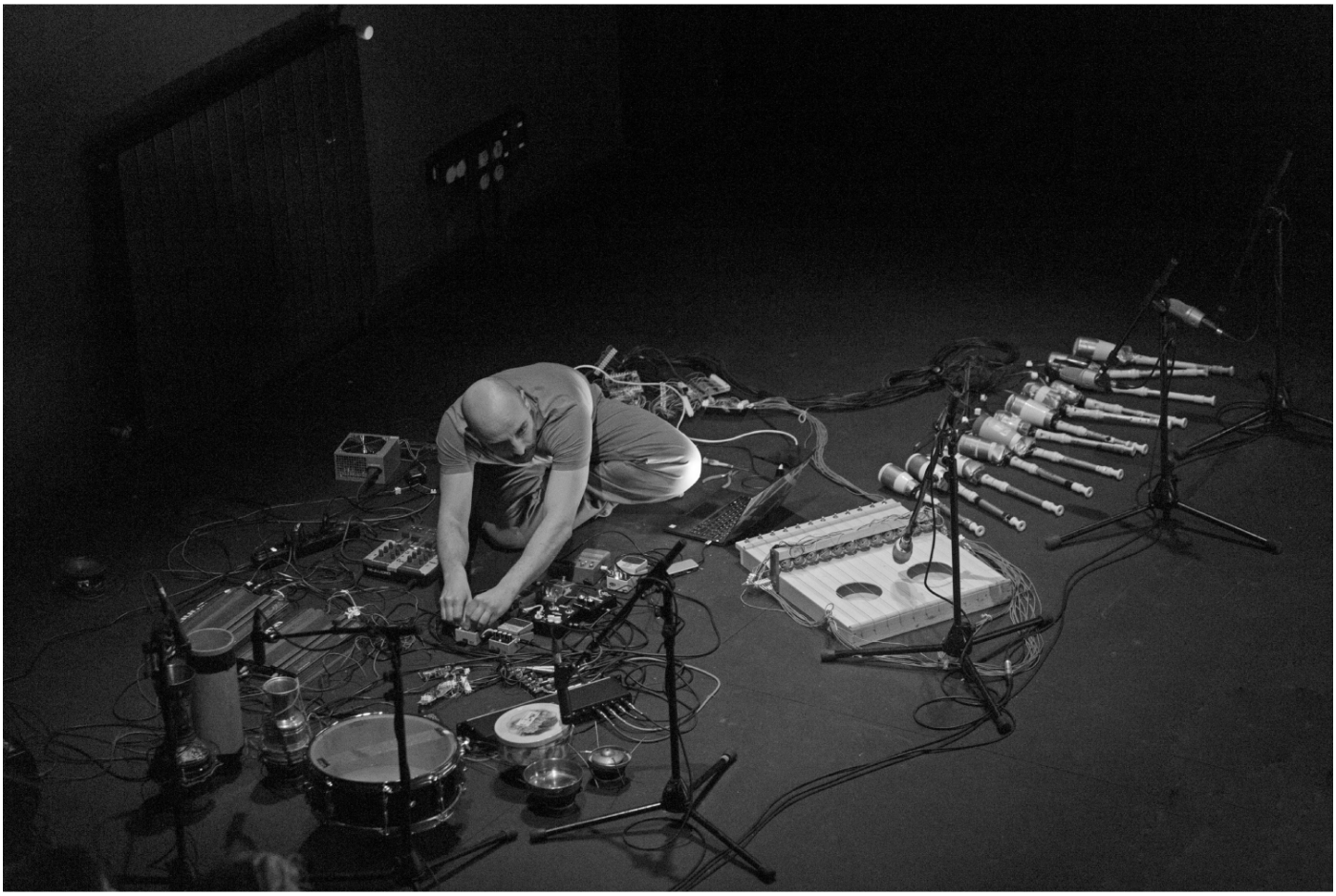


Figure 24.1
Live performance of *Ultraorbism* by Marcel.li Antúnez, featuring music by Andrea Valle, La Factoria d'Arts Escèniques de Banyoles. Banyoles, Spain. November 16, 2019. Photograph by Carles Rodríguez, 2019.

The work I do with code is expressed as sound essays with multiple layers of language: code, orality, and text. Live coding allows me to think out loud, with JITLib, I write and concatenate variables containing synthesis, sample granulation, OSC messages and MIDI connections.

Since 2022, I create artistic research on machine learning and artificial intelligence from a transhackfeminist perspective. I try other ways of making and put emphasis on small voice sound databases, “hacking with care.” I experiment with literature, poetry, and academic research on technology, art, and feminisms. My workflow consists of processing text databases and recomposing them with Markov chains. These recombined texts are interpreted with human voice, recorded, and analyzed with neural audio synthesis libraries. During the performance with SuperCollider, I detonate a dialogue of processes, resignifications, and interactions between the machine, interpreters, the public, and live coders.

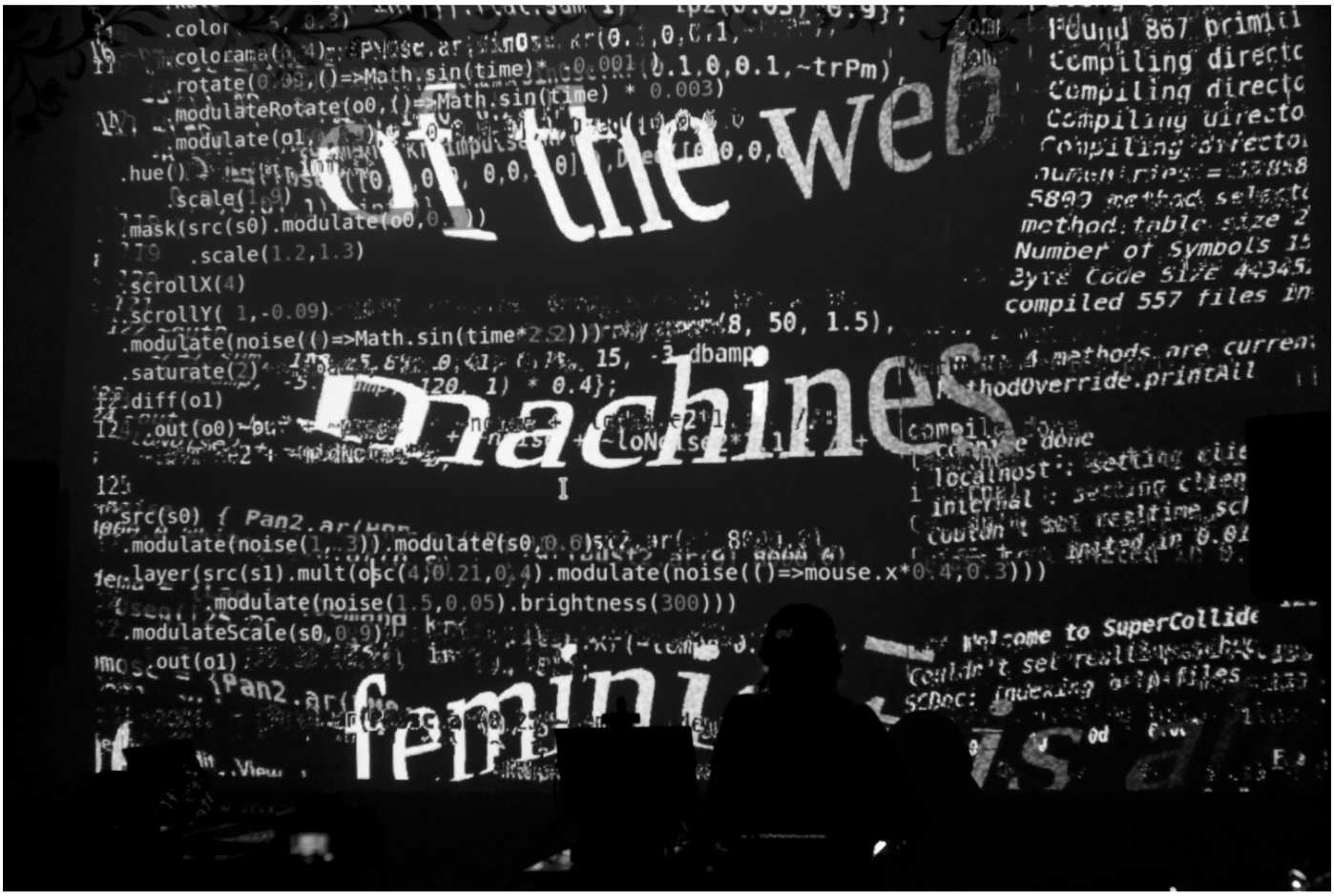


Figure 24.2
"Symbolic Machines" by Marianne Teixidó at Pumpumyachkan Asimtria 17. Cusco, Peru, 2021.

Photo by S.N.F.T. (aka Not).
(CC BY 4.0)

Neil Cosgrove

"Ideas are like fish. If you want to catch little fish, you can stay in the shallow water. But if you want to catch the big fish, you've got to go deeper" David Lynch (2016, p. 10).

In the late 1990s, Richard James introduced me to SuperCollider and ever since getting the first idea to actually work, that original rush moment got me hooked and kick-started a lifelong love affair. In those 20+ years, I've made many things, my biggest projects being LNX_Studio (a DAW, https://lnxstudio.sourceforge.io/) and World_World (a 2D games engine, https://github.com/neilcosgrove/Supercollider-World_World).

Because SuperCollider is interpreted, it's easy to experiment with ideas and filter the good from the bad. Being able to prototype and fail quickly enables you to spend your precious time on the big fish.

Knowing when to stop is also a skill worth developing. Many of my early projects suffered from feature bloat, to the point where they became unusable. Sometimes when I return to a project, I forget what a particular feature was designed for, which tells me I've dug too deep. Simple and focused tend to work best.

As long as it's open source, I'm never afraid of stealing other people's code. Often they will inspire you, save you time or teach you something new. It's always nice to give them a little nod in your code if you decide to share your projects with the world.

LNX_Studio is a DAW with many ideas. One is networking all its features: the ability for two or more people to collaboratively create a song, edit piano rolls, design sounds, and tweak effects in real time on separate computers. The network tools I created to do this directly influenced the LANdini project of Jascha Narveson and Dan Trueman (2013).

Part of the joy in developing your own musical toys is how creative it feels. You're designing the instruments that you and others use to create music, a doubly creative endeavor. In LNX_Studio, there are two modules, SC_Code and SC_Code_FX, where you can create your own instruments and effects with SuperCollider code. These modules have a suite of tools that allow you to generate and edit associated GUIs for parameters and sequence with piano rolls. For me, creating the tools that create the instruments that create the music is a triple creative hit.

Most important is to play and have fun. I've found so much joy in just experimenting and playing around. SuperCollider has been a big part of that.

Lynch, D. 2016. *Catching the Big Fish: Meditation, Consciousness, and Creativity.* Penguin.

Narveson, J., and D. Trueman. 2013. "LANdini: A Networking Utility for Wireless LAN-Based Laptop Ensembles." In *Proceedings of Sound, Music and Computing Conference*.

Shelly Knotts

SuperCollider was my first computer language. We met in 2005 through Scott Wilson's undergraduate module in Creative Computing at the University of Birmingham. At the time, I was interested in instrumental composition and saw SuperCollider as a way to expand the sound palette I was using as a composer. However, over time SuperCollider became central to my practice as a musician and brought me to new experiences. In 2010, I was studying for a masters in composition and started the group BiLE (Birmingham Laptop Ensemble) with other postgraduate composers at Birmingham. Following the model of The Hub, our pieces were largely frameworks for improvisation which included networked exchange of data and ideas. Through BiLE, I learned how to improvise and also wrote my first network music piece *XYZ* (2011) (see Knotts, 2013), which required performers to "fight" over control of parameters, building an ever-more-complex web of parameter mappings over the course of a performance. In 2012, my BiLE colleagues and I organised the Network Music Festival,

which has now had 4 editions (most recently in 2020). The NMF, more than anything else, was about fostering a community of musicians with an interest in the dynamics of networked collaboration, many of whom were SuperCollider users.

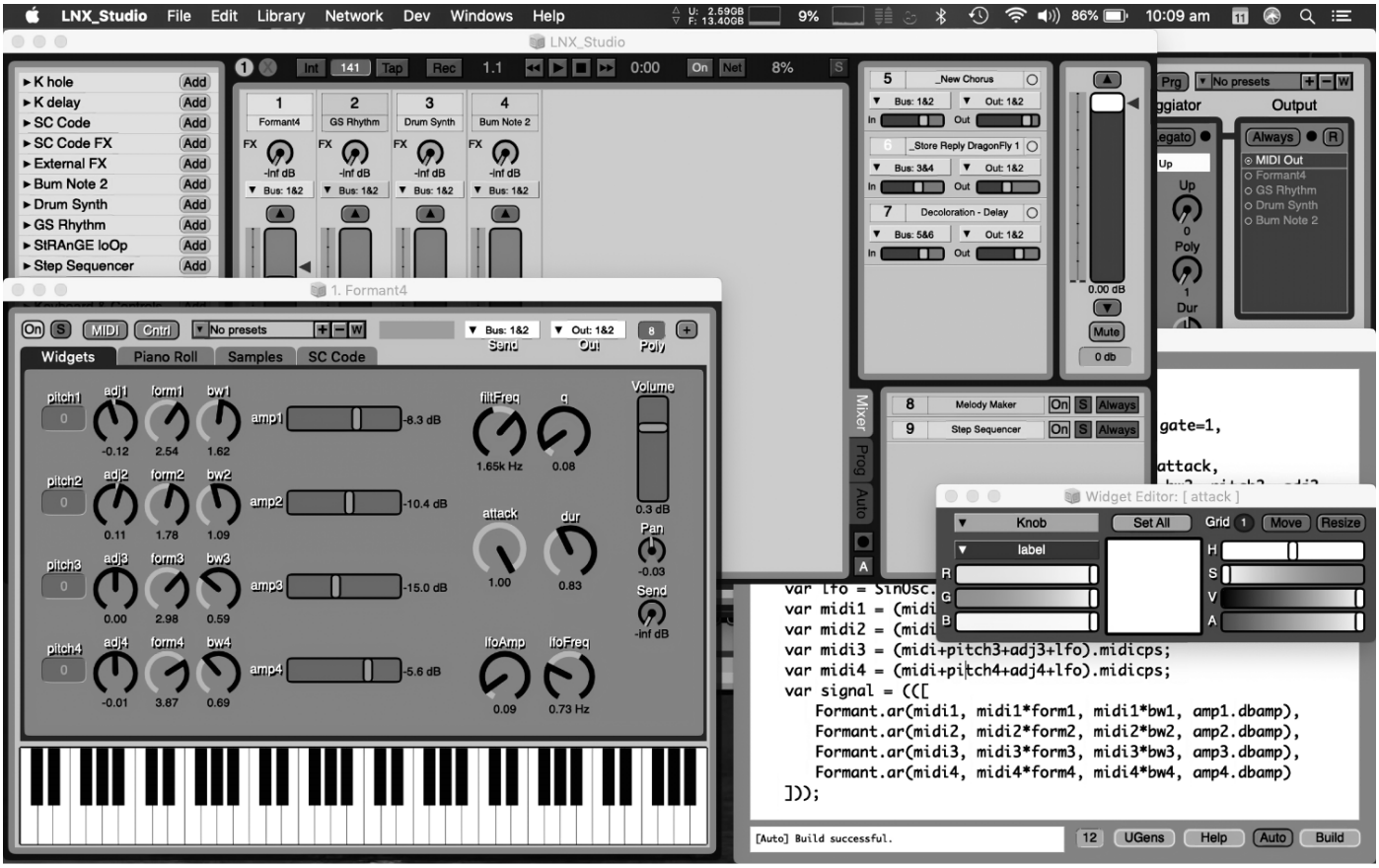


Figure 24.3
Screenshot from the LNX_Studio software.

SuperCollider challenged me again in 2012, when working on a project with jazz musicians. I realised the patches I was building for improvisation—which I controlled with simple hardware controllers—didn't give me the same fluency or breadth of musical responses as an acoustic instrument. I wasn't able to respond in the moment the way I wanted to. I had been aware of the practice of live coding since 2006, when the SuperCollider Symposium was held at the University of Birmingham, and included a Republic jam in the arts building foyer. I remember being excited by the possibility to write and change code in the moment, but it felt out of reach to me at the time. However, in 2012, motivated by my recent explorations of improvisation, I began learning to live code, at first meeting weekly with a fellow learner to jam before playing my first live coding gigs. In 2013, I played with Alo Allik at the inaugural live code festival in Karlsruhe—of course, in the moment of performance a quirk of SuperCollider caused the audio engine to crash and a quick restart was needed—one of many on stage crashes

to come! Through live coding in ProxySpace, I discovered a fluency with SuperCollider that I had missed in my earlier practices. I enjoyed discovering new sound worlds in the moment by writing complex and hard-to-read synth patches, building feedback loops, layering modulators, and surfing on the edges of chaos. In 2014, I took part in a residency for "nonidiomatic improvisation," where I spent two weekends jamming with acoustic improvisers. Here, I found once again that my fluency with SuperCollider was tested by the skills of my collaborators, but I enjoyed exploring strategies to meet them in the middle. Live coding led me toward many wonderful collaborations and experiences, including ALGOBABEZ (Knotts and Armitage, 2022) and the Algorave scene (Knotts and Collins, 2021), where I discovered dance music through SuperCollider code.

In recent years, I have been programming in other languages too, but I always come to SuperCollider first if I want to sketch an idea or explore a sound world. SuperCollider is still teaching me; through it, I have learned new skills, new sounds, new limits, and new experiences. The last 18 years of my musical life has been structured through my conversation with SuperCollider, and for that I am thankful.

Knotts, S. 2013. "A Portfolio of Compositions: Commentary." PhD diss., University of Birmingham.

Knotts, S., and J. Armitage. 2022. "ALGOBABEZ." In A. F. Blackwell, E. Cocker, G. Cox, A. McLean, and T. Magnusson, eds., *Live Coding: A User's Manual* (pp. 44–45). Cambridge, MA: MIT Press.

Knotts, S., and N. Collins. 2021. "AI-Lectronica: Music AI in Clubs and Studio Production." In E. Miranda, ed., *Handbook of Artificial Intelligence for Music* (pp. 849–871). Cham, Switzerland: Springer.

Juan Sebastián Lach

Since 2004, my musical practice has been intrinsically linked with SuperCollider, most of the work involving it in some way; it spans from tool/calculator—deriving materials, counting possibilities, outputting lists and tables, etc.—to apparatus central to the operation of instrumental and interactive music. As a general programming language, it allows multiple approaches; I tend a bit more toward the algorithmic-interactive than the sound synthesis side, though the more intertwined the two, the better.

The first SC-only work of that first year, *Victoria's Secret* (2004), was a generative piece in homage to and for Robert Ashley. It enacts a kind of Markov-chain news anchor (named after the operating system's voice, hence the title) mixing current reports of wars and conflicts with texts by Beethoven, William Blake, Bhagavat-Gita fragments, and others in a gradually self-annihilating process transitioning from the semantic to the phonetic accompanied by a microtonal quasi-granular basso continuo.

These pieces work as programs, but they're difficult to document. The ability to make stand-alone versions with SuperCollider facilitates collaborative work and interactive systems used by colleagues can be synced to make changes remotely for varying circumstances. Experiences in this category of pieces that stand out include *guajex*

(2013) with Diego Espinosa, *ad:am:an:ti:ne* (2014) with Carmina Escobar, and *Infrastruchor* (2015) with the ensemble Liminar. It also functions well when working with pieces by other composers, like the present-day realizations created for the electronics part of some of James Tenney's pieces—*Voices* (1983–1984) and *Critical Band* (1988)—adding features and adapting them to the contexts and requirements of interpreters.

SuperCollider has facilitated my interest in devices for improvisation growing out from pieces, reactive installations, generative materials and sonifications, especially dear to me, mathematical ones: I once modeled a topological "Hopf" fibration with quaternions which, when graphed and studied, permitted projecting parameters that undergirded a string quartet (Preimagen 2019). But perhaps it is most noteworthy to mention the language's extensibility as crucial to my practice. When I was doing doctoral research (Lach 2012), and stemming from musical needs, a library around sensory dissonance curves was developed as a toolbox (available as the `DissonanceLib` Quark) for working with microtonality and psychoacoustics. Its main purpose was converting spectra into automatically classified and formatted pitch sets used in works such as "strings" (2007), for guitar and computer, or the installation *Ahí Estése* (2013). It also permits working with the arithmetic of harmonic ratios, psychoacoustic units (converting an input amplitude into musical dynamics or balancing the instrumentation of a chord via the Bark scale, for instance), the rationalization and analysis of frequency lists, etc. Also implemented are ways of dividing intervals: harmonic means, ancient Greek katapyknosis (Chalmers 1992), the proportional systems of Mexican inventor Augusto Novaro (1951), as well as the harmonic metrics of Leonhard Euler (1739), Clarence Barlow (2012), and Tenney (2014).

Lastly, I should mention SuperCollider as a significant pedagogical tool beyond teaching someone how to program and compose. It's helpful for demonstrating (and visualizing) sound phenomena, acoustics, music theory, and soni/musification, as well as the dynamics of open-source projects and communities. At this point in life, I couldn't do without such a programming language. It is a storehouse, which can be activated and expanded, of many of my musical experiences and skills.

Barlow, C. 2012. *On Musiquantics*. Mainz, Germany: Musicological Institute/Musikinformatik & Medientechnik of the University of Mainz. Accessed February 2, 2023, from http://clarlow.org/wp-content/uploads/2016/10/On-MusiquanticsA4.pdf.

Chalmers, J. 1992. *Divisions of the Tetrachord*. Lebanon, NH: Frog Peak Music. Accessed February 2, 2023, from http://eamusic.dartmouth.edu/%7elarry/published_articles/divisions_of_the_tetrachord/index.html.

Euler, L. 1739. *Tentamen novae theoriae musicae ex certissismis harmoniae principiis dilucide expositae*. St. Petersburg Academy.

Lach, J. S. 2012. "Harmonic Duality: From Interval Ratios and Pitch Distance to Spectra and Sensory Dissonance." PhD diss., Leiden University, Academy of Creative and Performing Arts, Leiden, Netherlands. Accessed February 2, 2023, from https://hdl.handle.net/1887/20291.

Novaro, A. 1951. *El sistema natural de la música*. Mexico City: author's edition. Accessed February 2, 2023 from https://www.anaphoria.com/novaro51.pdf.

Tenney, J. 2014. "John Cage and the Theory of Harmony (1983)." In J. Tenney, L. Polansky, L. Pratt, R. Wannamaker, and M. Winter, eds., *From Scratch: Writings in Music Theory*. Chicago: University of Illinois Press. accessed February 2, 2023, from http://www.plainsound.org/pdfs/JC&ToH.pdf.

Mileece

https://www.mileece.is/

"I involved myself with computers because of music.

"I remember the moment I realized I could use them to create compositions with virtual orchestra and generative sounds, without having to work with anyone else!

"Albeit these sound like the words of an agoraphobe, this realization was a world of potential suddenly opening in my mind; the computer was an unexplored instrument that could offer the possibility to create music not previously attainable and I could do it without the limitations of traditional musical rules.

"I quickly got into learning Max/MSP and SuperCollider, as programming languages like SuperCollider allow you to create personalized or project-based interfaces, as well as your own functions and data sets; you can thrive from an efficient platform designed by your own creative logic, specifically for your structural requirements.

"Prebuilt software programs such as Logic and Pro Tools, although undeniably often very useful and in many cases much less time consuming, constrain the user to navigate within a limited set of predesigned possibilities. With programming languages, the better you learn them, the more articulate you become and the interface between mind and music dissolves, in some ways like any traditional instrument. But there is also something unique to programmed music that no other instrument has; it can be both performer and composer, and interact in an entirely new way. Low-level artificial intelligence as a navigational or divinatory tool (depending on how you wish to perceive it) allows for structures to generate autonomously much in the same way as a plant or any other living entity would. It starts with a set of principles and grows.

"Like this, simple musical structures return to being given room to express living relationships; tempo and rhythm are formed as nonrepetitive periodical patterns iterated in a cyclical form and what are otherwise mainly static elements in modern Western music, become algorithmically generated in an ongoing series of chance operations."

I put this excerpt in quotations, as it's edited from an entry I wrote back in October 2002, called "Why Computer Music." I can pick it apart for its clumsiness, but I think it conveys the spirit of our collective love affair with SC with a youthful gusto (and reveals my long and joyful abuse of the semicolon). In any case, I wrote it to explain, or at least contextualize, the release of *Formations* in early 2003, an album primarily composed of recordings of SC patches based on patterns in nature, purposefully iterated with a minimal timbre impossible to make naturally, and designed for multichannel

diffusion systems. It was released at a time when women in music, and especially coded music, were both as rare as undernourished, yet it was well received enough to help me track toward the next step in my ultimate goal: creating biologically driven electronic music. Segueing from algorithms, I began creating "gestural interfaces," sensor-based instruments usually made out of found natural objects and/or antique materials, mapped to generative sound modules in SC.

In 2005, which I remember felt like decades later, I presented the first interactive installation using PiP (People's Interface for Plants), a platform designed to transform the bioemissions of people and plants into harmonic, generative soundscapes. Still using SC to this day, PiP detects and relays biowaveforms in a process that is, to me at least, pleasing to listen to and which I call "aesthetic sonification."

In essence, it's safe to say SC, and my early forays into the world, allowed me to create, underpin the wider scope of projects I still design today, all with the unabashedly grandiose desire to promote ecology through art and technology and help inspire a culture of eco-citizenry and planetary well-being. Technology is our dream; Earth is truth.

Sam Pluta

www.sampluta.com

I wrote my first musical works with SuperCollider 2 in 2003 and moved to SuperCollider Server later that year. I have been using it ever since in tasks as trivial and everyday as simple math calculations up to controlling audiovisual installations and composing algorithmic works for ensembles. See *American Idols* (http://sampluta.com/americanIdols.html), an installation for feedback guitars and televisions, and *Seven Systems* (http://sampluta.com/compositionSevenSystems.html), a concert work for two pianos, two percussion, and electronics.

What I love about this language is that through a combination of robust DSP, myriad data structures, and a quirky object-oriented design, it gives the musician the infrastructure to express all kinds of music, from noise music to electronic dance music to algorithmic composition. The language makes no assumptions about what music is, but rather, it offers the user an open environment to find their own paths to expression through exploration and discovery.

This ethos is at play in my large-scale performance software environment, the Live Modular Instrument (https://github.com/spluta/LiveModularInstrument), which has been my main performance tool for over a decade and was the topic of my doctoral dissertation (Pluta 2012). This software instrument provides me with a data structure/interface that allows easy and rapid access to many sound synthesis and processing algorithms. Because SuperCollider Server can run on multiple central

processing units (CPUs) at once, the current version lets me run almost every plug-in I have written over the past two decades at the same time and have access to each of these plug-ins at the tips of my fingers.

I have performed on my software instrument with jazz musicians, noise artists, classical instrumentalists, analog synth players, and drummers, and have recorded and released almost 20 records that feature it. My software is set up so that I can control it with any OSC, MIDI, or HID hardware, and I can even fold in any other software or hardware on or connected to my computer, allowing me to play a synthesizer in Reaper as well as my Eurorack synthesizer through a sound-processing algorithm I might have written yesterday or up to twenty years ago, or maybe both at the same time. All this is possible because of the incredible foresight James McCartney had in designing such a musical, expressive, and open-ended programming language.

Many thanks to the incredible SuperCollider community for their support and dialogue over the past two decades. I have learned so much from this caring group and could never have made my music without them.

Pluta, S. 2012. "Laptop Improvisation in a Multi-Dimensional Space." Doctoral dissertation, Columbia University, New York, https://academiccommons.columbia.edu/doi/10.7916/D8S188KG

Ann Warde

zsonics.org

SuperCollider routines underlie my experimental projects *Hidden Encounters* and *Doubtful Sound*. The unpredictable surface layer of these compositions is supported by intersecting and inter-related patterned structures. Listeners and performers are invited to investigate how a seemingly indeterminate musical experience might nonetheless be shaped by a complex, dynamic, hidden framework.

Hidden Encounters is an interactive sound installation which has evolved via multiple iterations. In its first rendering, a listener crossing the beam of a *hidden* infrasonic range finder triggered the playback of a sound file, which was then played back a second time with its playback speed altered. A second realization combined this SuperCollider code and Rebecca Fiebrink's Wekinator machine learning software. The presence and movement of listeners, detected one-dimensionally by the range finder, determined changes in the playback rate and amplitude of sound files. This took time: listeners were requested to listen inquisitively. A third realization's SuperCollider code evolved further, focusing on running iteratively overlapping multiple copies of a single routine that played back selected files at altered amplitudes and playback rates. This installation benefited from a combined set of ceiling-mounted and free-standing loudspeakers in a large gallery space. However, although the computer-controlled sound file playback system was dynamically changing, the infrasonic range finder's

independent alteration of additional, very short sound files (in response to highly localized movement by listeners) was active in only a small region of the gallery. *Hidden Encounter*'s SuperCollider routine is relatively simple, but the precise ranges within which indeterminate values for sound playback alterations were chosen proved crucial to the installation's musical result. The sound files are field recordings of unaltered animal vocalizations, and extreme settings for playback rate and amplitude ranges proved to be particularly useful (Figure 24.4).

Doubtful Sound is an interactive performance system; further information about this project is in the Networks issue of the Orpheus Institute's online journal *Echo* (https://echo.orpheusinstituut.be/article/doubtful-sound). An undirected network region from the early graphic score *Cassiopeia* by George Cacioppo determines the playback order of a set of 12 files. The playback rate and amplitude of segments of these sound files are altered according to values chosen sequentially within a directed network of dolphin social interactions (collected and constructed by sustainability scientist David Lusseau at Doubtful Sound, New Zealand). The values assigned to each network are initialized at the start of the playback routine, and as long as an instance of the routine is active, the sequence of its values does not change. A performance may contain multiple instances of the playback routine. The system also "listens" during a performance for changes in the amplitude of sound captured at half-second intervals. Higher amplitudes produce longer lengths of time between segments of sound files whose playback rates and amplitudes are altered. The result is a potentially nearly, but not clearly, predictable situation. The music asks: In the midst of the system's surface unpredictability, how might we experience *Cassiopeia*'s sonic network structure (as built by its composer), and the social structure of the Doubtful Sound dolphins?

Anna Xambó Sedó

annaxambo.me

SuperCollider is like a good friendship that resists the passing of time. It has been critically influential upon my artistic practice and finding my voice in designing networked algorithmic music performance systems. The potential for chance, collectiveness, indeterminacy, openness, and uniqueness makes SC one-of-a-kind.

In 2006, I went to see the presentation of SuperColliderAU by Gerard Roma at the first SuperCollider Symposium at the University of Birmingham, UK. Getting to know the first generation of SC artist-developers was very inspiring. I realized the fine line between being an artist and a developer with the design of tangible interfaces for music performance using SC. My SC mentor, Gerard, and I presented a sound editor with a tangible interface at the 2007 SC Symposium in The Hague. In 2012, I continued the exploration during my PhD at the Open University in the UK developing the side project

Soundscape DJ, again together with Gerard, which won the Warp Records prize at the Music Tech Fest in London. Algorithmic sound-based music has been my interest since then, as found later in my live coding practice.

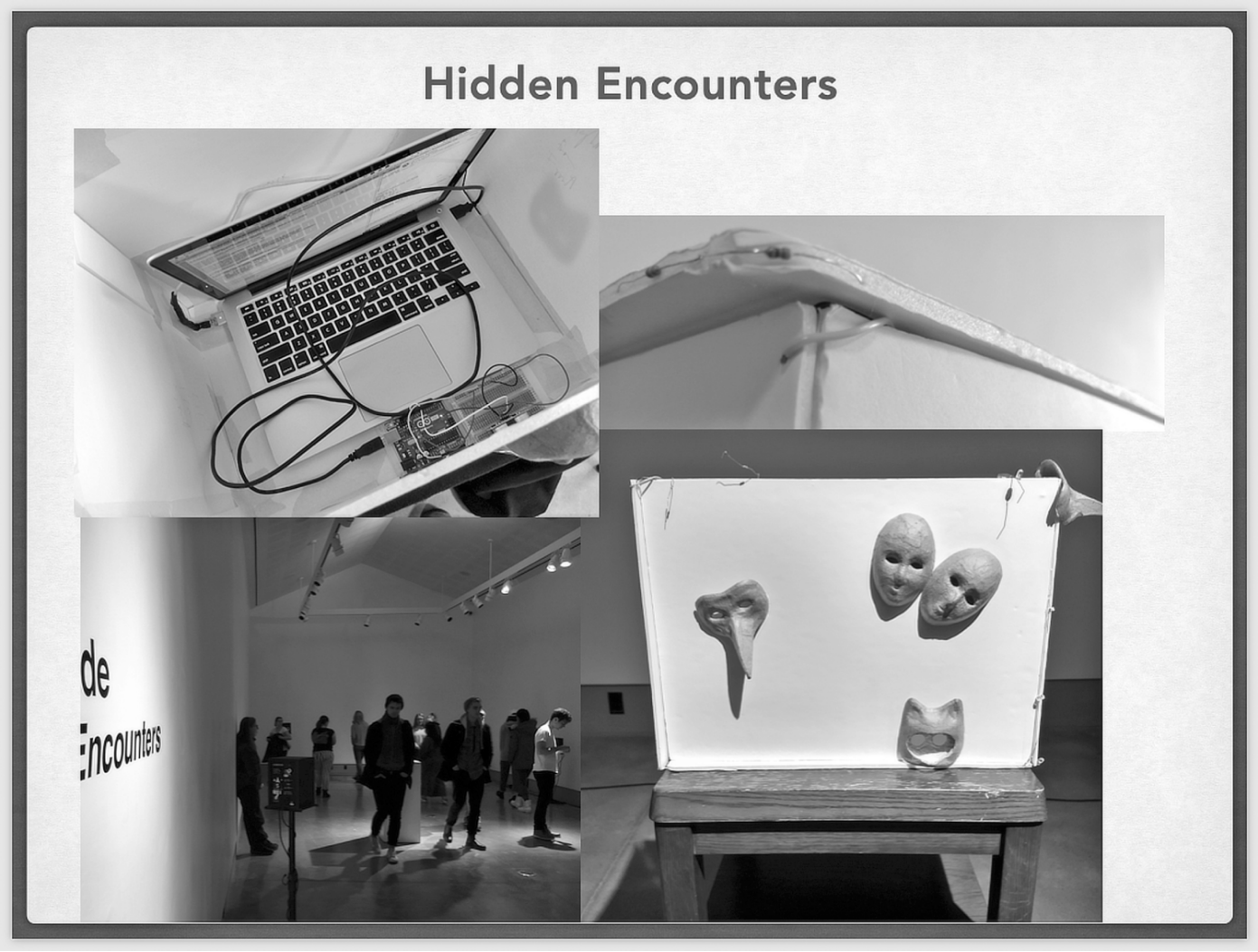


Figure 24.4
Hidden Encounters sound installation, Ann Warde.

Upper left: Inside the box, The Global Composition 2018, Dieburg, Germany
Upper right: Top edge of the box, The Global Composition 2018, Dieburg, Germany
Lower left: Visitors, Art and Media Lab, Isabel Bader Center, Queens University, Ontario, Canada
Lower right: Box, Art and Media Lab, Isabel Bader Center, Queens University, Ontario, Canada
All photos are by Ann Warde.

Live coding was a tipping point. It has allowed me to try improvisation on stage, as a solo artist as well as part of acoustic and electronic ensembles: in 2012 with Pulso, an electronic music duo with Gerard using a self-built collaborative live coding interface; in 2013, as a solo performer in a multichannel live coding concert hosted by Phonos in Barcelona; in 2018, as part of the Female Laptop Orchestra performing a hybrid

network music performance at SARC in Queen's University Belfast, with 10 women performers on site and remotely; and in 2019, as part of the Trondheim Electroacoustic Music Performance at the Web Audio Conference.

Another tipping point was to accept the "developer" hat as part of my practice. In 2016, I started developing the SC extension MIRLC (https://github.com/axambo/MIRLC) for music information retrieval in live coding (Xambó, Lerch, and Freeman 2018) that was used in collaborations such as a duo with the visualist Anna Weisling, performing at NIME 2017 and 2018. In 2020, I explored machine learning in live coding (Xambó, Roma, Roig, and Solaz. 2021) with my self-built SC extension MIRLCa (https://github.com/mirlca), part of a project that was funded by the Human Data Interaction network (https://hdi-network.org); a new version is under development. I have tried MIRLCa both as a solo performer and as part of ensembles (Xambó 2023), such as the Dirty Electronics Ensemble and Jon.Ogara, documented in a live music album in 2021, or a duo with Visda Goudarzi investigating live coding and participatory audience in a network music performance at NIME 2021.

SC has allowed me to build my tools for music-making and to dive into new algorithms creatively. In my journey, I have enjoyed the company of a great community that likes to share and try each other's code and ideas. As part of the recently created women's live coding collective livecoderA (https://livecodera.glitch.me), I am certain there is a bright future for women in live coding.

Xambó, A. 2023. "Discovering Creative Commons Sounds in Live Coding." *Organised Sound*, 28(2): 276–289.

Xambó, A., A. Lerch, and J. Freeman. 2018. "Music Information Retrieval in Live Coding: A Theoretical Framework." *Computer Music Journal*, 42(4): 9–25.

Xambó, A., G. Roma, S. Roig, and E. Solaz. 2021. "Live Coding with the Cloud and a Virtual Agent." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Shanghai: NYU Shanghai.

25 Machine Learning in SuperCollider

Chris Kiefer and Shelly Knotts

25.1 Introduction

Machine learning (ML) techniques offer musicians some weird, wonderful, and sometimes profound ways of making sound, music and instruments. They can be deeply rewarding creative tools (Fiebrink, 2017). ML is an area within artificial intelligence (AI). The terms *AI* and *ML* (Russel and Norvig, 2020) are often heard together, but what is the difference between them? AI tends to describe modeling of human sensing and cognition using computational processes, sometimes with the grand aim of creating Artificial General Intelligence (Goertzel and Pennachin 2007), to make machines that are humanlike in their behavior. ML sits within the wider field of AI and is concerned with making machines that can perform specific tasks through varied processes of training or optimization. Sometimes these machines use biological concepts based on our understanding of human and animal brains.

Typically, when we create new software, we engineer it manually by writing code. ML gives us methods for creating software by training instead. The training process allows us to create software that can perform complex tasks that would be difficult or impossible to create manually. This makes ML particularly useful for addressing challenging problems in computer music, for example differentiating between sounds or generating new music in a particular style. Training begins with an untrained model. A model is a sort of system that is capable of arbitrary computation. It has a large number of parameters which can be tuned through training such that the model performs a particular function. ML models may have thousands or billions of these parameters. A typical model is a *neural network*, which is a simplified model of the neurons in a brain.

Training methods roughly fall into two categories: *supervised* and *unsupervised*. With supervised methods, we start by making a *training set*. This is a collection of pairs or inputs with corresponding outputs. For example, if we want to train a model to add reverberation to audio, we could collect training pairs where the input is a dry source sound and the output is a recording of the same sound from within a resonant space. Having collected this data, we can then run *training*, which will tune the

parameters of the model to try and match the process that transforms input into output, based on the examples from the training set. The training will rarely be 100 percent successful, but we hope that it will be able to make a good approximation of how the resonant space affects the dry sound. Once our model is trained, we can then run this model on new sounds, and it will add reverb to them with the characteristics of the resonant space where we recorded the training set.

It's likely that we will run the model with sounds that were not within the original training data. If the model works successfully to add reverb to these unseen sounds, then we can say that it has *generalized* well. Because the training process is unlikely to be 100 percent accurate, and because the model will be run on inputs beyond the original training set, there may be some unexpected ways in which the reverb behaves. Actually, some of these unexpected behaviors may sound good, or at least coherent in some way with the original training examples. These unexpected behaviors are often interesting creatively; ML models are excellent sources of serendipity, giving us surprising and interesting creative results. Sometimes they go wrong in spectacular ways. In fact, the way in which we train models is really a form of composition. Different combinations of training materials will give different creative results; we can experiment with varied pairs of inputs and outputs to create diverse models. Models are malleable; they can be adjusted, retrained, and recombined to make new models and new sounds.

In *unsupervised* training, we do not pair data into inputs and outputs; instead, we show the model of all our data, and training optimizes this model so it can respond to the underlying structure and relationships within the data, for example, to reflect differences between sounds. Unsupervised training is often used to make models that are *generative,* meaning that they are used to create new content. For example, we can take a collection of our favorite songs and train a model that will generate new music based on these songs. Generative models offer varying degrees of control over the content that they produce, and as with supervised learning, we can be creative in the selection of the data in the training set. For example, if we train a model with a combination of arias, drum and bass, and recordings of cows, perhaps we could pioneer a new genre of farmyard operatic breakbeats (FOBcore).

In this quick introduction, we've learned some key terms: *models, training, training sets,* and *generalization*. Now we can learn how ML can be a valuable creative material for sound designers, musicians, and instrument designers and begin to explore ML through practical applications in SuperCollider (SC).

25.2 Creative Machine Learning in Sound and Music

Creative machine learning is becoming ever more accessible to computer musicians as new tools and methods become available that reduce the complexity of the programming

task (McCallum and Grierson, 2020; Roberts et al., 2019; Bernado et al., 2020), and as advances in computer hardware enables us to process data at faster rates on our home machines. In SuperCollider, various UGens and extension libraries are available to facilitate machine learning tasks. At the end of the chapter, we also showcase how to interact with models in other languages, which opens up a full scope of machine learning tasks while still harnessing the live synthesis power of the SuperCollider engine.

Machine learning applications can process huge data sets (Collins, 2016), finding patterns that might not be obvious through manual analysis techniques, which can then be harnessed for creative use. This gives us the possibility to generate endless recombinations and reconfigurations of musical materials, producing new and unexpected ideas. MASOM (Tatar and Pasquier, 2017) is an improvisation system that uses self-organizing maps (SOMs) and neural networks to analyze musical forms. In REVIVE (Tatar et al., 2018), the system is trained on a large corpus of experimental and electronic music and responds to the human performers with sonic gestures which retain characteristics of the corpus material.

In combination with machine listening techniques, we can create our own improvisation partners, that can learn our musical style and push us in new directions—and may be less troublesome than our human collaborators to boot (Knotts and Collins, 2020). Pachet's Continuator (2003) utilizes Markov models to imitate musical styles, generating new sequences in real time which expand on the user's input. IRCAM's OMax (Lévy, Bloch, and Assayag 2012) uses the Factor Oracle algorithm for style modeling and real-time music generation, recombining past musical figures and challenging human performers to explore new directions.

Machine learning methods can speed up mapping processes, creating more complex mappings in less time and more intuitively than manual coding can achieve (Kiefer, 2012). Fiebrink's Wekinator (2009) allows musicians and instrument designers to train machine learning models on the fly with custom mappings using a supervised learning method. The article "The Machine Is Learning" (Baalman 2020) critically highlights the labor of training machine learning models by making the dialogue between machine and human in the process of mapping gestures to sound a theatrical performance.

Beyond that, we can construct parameter spaces for creative exploration, presenting a continuum of possible parameter combinations in navigable ways, optimising or expanding our creative processes by saving us the need to test endless recombinations. R-VAE (Vigliensoni et al., 2022) uses a customized variational autoencoder (VAE), a more recent learning model, to encode musical rhythms. R-VAE focuses on small data sets; a model can be built with as few as twelve rhythmic patterns. A visualization representing the model's internal *latent space* allows easy exploration of rhythmic interpolations. Another VAE model, RAVE (Caillon and Esling, 2021), provides fast

and high-quality waveform synthesis, enabling real-time timbre transfer. (Associated UGens for this model are now available for SuperCollider at https://github.com/victor-shepardson/rave-supercollider.)

We can also automate processes that might otherwise require human input. These methods can reduce our human coding workloads and decision-making, creating more cognitive space and time for the more creative tasks. In SuperCollider, Xambó's MIRLCa system (Xambó et al., 2021) enables live coders to pretrain a model on the performer's preferences of musical examples from FreeSound.org. During a performance, the model filters potential sound sources retrieved from FreeSound.org according to trained preferences. Live Coding is a practice with a typically high cognitive load and MIRLCa frees the live-coding performer to concentrate on other tasks than sound selection during performances.

For further examples of creative uses for machine learning, Miranda (2021) serves as an extensive guide to current creative exploration with AI in sound and music, and a deep exploration from a practitioner's perspective can be found in Fiebrink and Sonami (2020).

25.3 Examples

In the main section of this chapter, we'll learn about ML in SuperCollider through practical examples. We will explore supervised and unsupervised learning, beginning with simpler statistical models, before moving to more conventional neural networks, and finally an example of deep learning (Bengio et al, 2017). With all these examples, you are encouraged to creatively experiment with the training and inferences; you could consider them all as basic starting points that can be developed and expanded into bigger projects.

25.3.1 A Super Simple Markov Sequence Generator

Markov models are stochastic methods that can generate new sequences based on the probability of the next value in relation to the previous *n* values or states. This makes them very good at generating new sequences from a relatively small data set that will strongly resemble the given data, and yet surprise us with new variations. They can be used creatively, for example, to generate new variations on a given melodic sequence.

In this example, we generate two Markov models, one for pitch sequences and one for rhythmic values, using the main vocal melody of Cyndi Lauper's "Girls Just Wanna Have Fun" as the input data. We can then generate new sequences of notes with similar melodic characteristics to Lauper's original. We use a variable-order Markov model, PPMC; our order value in the PPMC model defines how many previous states the model considers when generating the next value. Higher-order numbers will produce

sequences that are more congruent with the original data set but less likely to generate novel and surprising sequences.

This is a relatively simple and cheap method for unsupervised learning that can have a wide range of uses in the creation of music; however, as the output is localized to predicting the next state based on the current state and recent past, they cannot deduce, predict, or generate longer-term musical forms.

We'll now describe the code for a simple melodic sequence generator using SuperCollider's PPMC UGen, trained on "Girls Just Wanna Have Fun." You can find PPMC in the SCMIR Library (Collins, 2011), which can be downloaded from: https://composerprogrammer.com/code/SCMIR.zip

In this discussion, we'll look at the details of the code, section by section.

In figure 25.1, we first load our data from two CSV files and convert it to integers. These contain the pitch sequence and inter-onset interval (IOI) sequence from the vocal line from Lauper's song. We generate two instances of the PPMC model and train them using our data. You can play around with the order value to generate sequences with more or less congruence to the original.

```
a = CSVFileReader.read("girls_melody.sc".resolveRelative).   flatte
n.asInteger;
b = CSVFileReader.read("girls_rhythm.sc".resolveRelative).   flatte
n.asInteger;
~mel = PPMC(3);
~mel.train(a)
~rhy = PPMC(3);
~rhy.train(b)
```

Figure 25.1
Loading data from two CSV files and training two PPMC models.

It was found through trial and error that an order value of 3 produced the most pleasingly Lauper-esque results with scope for novel sequences. As Markov models are very fast to train, retraining the model with a new order value is fairly trivial.

```
SynthDef(\a, {|freq = 60, amp = 1, fb=1|
   var snd, env;

   snd = Decay2.ar(SinOscFB.ar(freq.midicps * [1, 1.01], fb, 3)   .
tanh,0.005, 0.004).tanh;
   snd = FreeVerb.ar(snd);
   env = EnvGen.kr(Env.perc, doneAction: Done.freeSelf);
```

```
    Out.ar(0, snd * env * amp);
}).add;

(
var lastvals = [[a[0]],[b[0]]];

Tdef(\gen, {

    loop {
        var nextmel = ~mel.generate(lastvals[0].reverse);
        var nextrhy = ~rhy.generate(lastvals[1].reverse);

        Synth(\a, [\freq, nextmel]);
        (nextrhy*0.25).wait;

        lastvals[0] = lastvals[0].addFirst(nextmel);
        lastvals[1] = lastvals[1].addFirst(nextrhy);

        if(lastvals[0].size>3, {lastvals[0].pop; lastvals[1]   .po
p;});
    }
}).play;
)
```

Figure 25.2
Generating values for pitch and IOI and playing them in a loop.

To generate our new sequences, we make a loop that generates values for pitch and IOI, saves them to an array, and then plays the synth using our generated values.

Next, we initialize an array where we will store our new values. We insert the first values from our training data as arbitrary values to generate our first state. We then use the `generate` method of PPMC on our trained models, using our array data as the input to generate a new pitch and a new IOI. We play the synth using the newly generated pitch value and `wait` for the generated IOI time. The IOI is scaled by 0.25 as we trained the model on whole numbers, and our midi pitch value is translated to cps in the synthdef. We then store our generated values in the `lastvals` array to be used in the `generate` method on the next loop. Once our array has more than 3 values stored, we start dropping old values from the array, as we are never looking at more than the previous 3 values to generate the next state.

25.3.2 Remixing a Sound File Using Self-Organising Maps

Like Markov models, a SOM is used for unsupervised learning (i.e., asking the algorithm to deduce patterns from a given data set). SOMs are used to reduce complex data sets with many dimensions to fewer dimensions. This helps us find patterns and interesting features in the data that we can use creatively. SOMs can be used as clustering tools, as data will be arranged with relative similarity, so that more similar data are closer together and vice versa.

In our example, we are using the SARDNET UGen. You can find SARDNET in the SCMIR Library. We'll also use this library to extract feature data from our sound file.

SARDNET is a type of SOM that is able to handle sequences. This is useful in musical applications, as we are most often dealing with temporal data sets. In our SARDNET example we extract feature data from a sound file, segment it, and reorganize it by the relative similarity of the feature vectors contained in each segment. This allows us to navigate the sound file in a different and creative way. We'll be creating a map of Beyonce's "Hold Up," but you can replace this with your own sound file.

We start by extracting feature data from the sound file using SuperCollider's SCMIR Library. Chapter 15 describes this process in more depth, but in this context, it gives us lots of data about the sound material in our file to play with in our model.

In the training process, a SOM does the following: it generates "representatives" with random values, chooses an input data point at random, finds the representative which is the closest match to the chosen data point, adjusts the values of that representative to be closer to the input data, and adjusts the neighbor representatives to be closer to that data. Over the iterations of the training process, this builds a map of representatives which closely resembles the shape and clustering of our input data.

Now we'll look at the details of this code, section by section.

```
f = SCMIRAudioFile("Beyonce_HoldUp.wav".resolveRelative,   [[MFCC,
 13], [Chromagram, 12]]); //, Loudness, SensoryDissonance,   SpecCe
ntroid, SpecPcile]);
// we're just looking at the pitch trajectory in this example but w
e could include other features

{f.extractFeatures();}.fork; //wait for me to finish

(
var hopsize = 1024;
var frames_per_sec = s.sampleRate / hopsize;
var siz = frames_per_sec* f.numfeatures;

~num_samples = f.numframes.div(frames_per_sec) //dividing it by   f
```

```
rames per second gives us 221 entries—there are approximately   43
 frames per second so we end up with 1 vector of 43 data   points p
er second of the track.

~reps = Array.fill(f.numframes.div(frames_per_sec), {|i|

}) // the feature data is clumped into sets of 43 frames of   featu
re data values.
)
```

Figure 25.3
Extract feature data.

In our example, we start by extracting feature data from a sound file using `SCMIRAudioFile`. We specify the features that we want to extract. In this example, we are just using pitch data for illustration. After we extract the feature data, we prepare it for training the model by segmenting it into 1-s chunks. For our sound file, this gives us 221 data vectors, each with 43 time points. Each time point is described by 25 data values of features.

We then generate a `SARDNET` instance with 100 representatives and train it using our 221 data vectors. (See figure 25.4.)

```
a=SARDNET(f.numfeatures, 100, 300, 0.5, 0.5);

~reps.do{|array| a.train(array);}; // we're training the model   wi
th our arrays of data.

~results = ~reps.collect{|array| a.test(array).collect{|val|   val
[0]}}; // testing each second of the song against the   SARDNET ma
p. Each input sample produces a sequence of   representatives. .tes
t posts the representative which is   closest to the input data.
//each result is a sequence of 43 representatives each representing
one value (set) in the input vector: e.g. [16, 18, 17, 19, 20, 15,
 14, 21, 13, 22, 25, 12, 11, 60, 61, 23, 10, 24, 26, 27, 59, 30, 5
8, 31, 33, 32, 28, 9, 7, 6, 8, 5, 4, 3, 2, 1, 0, 29, 34, 35, 36, 3
7, 38]
```

Figure 25.4
Generate and train a `SARDNET` instance, then test each second of the song against the SARDNET map.

We can test our original data against the `SARDNET` map that we generated. This gives us a vector of representatives for each input vector. (See figure 25.5.)

```
~data = Array.fill(~num_samples,{|i| Array.fill(~num_samples,{|j|
(~results[i]-~results[j]).abs.sum});}) // calculate the   manhatten
distances between results for each input sample

~data_ordered = ~data.collect{|item|
   var sorted = item.deepCopy.sort;
   sorted.collect{|itm| item.indexOfEqual(itm);};
} // reorder the data to give the order of similarity from   each p
oint in the soundfile.
```

Figure 25.5
Calculate distances between the vectors and record according to similarity.

In this example, we want to reorder the sound file by the vector similarity of our segments. We can think of this as gestures, although our segmentation is entirely arbitrary here. First, we find the distances between the vectors, comparing each of our 221 result vectors to each other result vector and then ordering the data vectors by distance. This gives us a set of 221 orderings of segments, using each segment of the sound file as a starting point. (See figure 25.6.)

```
b = Buffer.read(s, "Beyonce_HoldUp.wav".resolveRelative);

SynthDef(\bufl, {| out = 0, bufnum = 0, sustain = 1, start = 0,   a
mp = 1|
   var env, buf, snd;
   env = EnvGen.kr(Env.linen(0.01, sustain, 0.01), doneAction:2);
   buf = PlayBuf.ar(2, bufnum, BufRateScale.kr(bufnum), 1, start
 * 44100 * BufDur.kr(bufnum), doneAction: Done.freeSelf);
   snd = buf;
   Out.ar(out, env * snd * amp)
}).add

SkipJack({defer {~cursor=QtGUI.cursorPosition;}}, 0.1);
~windowX = Window.screenBounds.extent.x; // calculate the screen
 width to later scale our mouseX value.
~x_pos = (((~cursor.x/~windowX) * (~num_samples-1))).floor   .asInt
eger.postln;

(
Pbindef(\a,
   \dur, 1,
   \legato, 0.99,
```

```
    \start, Pseq(~data_ordered[~x_pos]/(~num_samples-1), inf),
    \bufnum, b, // scrolling through the buffer
\instrument, \bufl
).play;

Tdef(\x, {
    var old_xpos = 0;
    loop {
    1.wait;
    ~x_pos = ((~cursor.x/~windowX) * (~num_samples-1)).floor   .asIn
teger; // extract the current x position of the mouse   and scale t
o 0-221
    if (~x_pos!=old_xpos, {
    ("new x position:" ++~x_pos).postln
    Pbindef(\a, \start, Pseq(~data_ordered[~x_pos]/(~num_samples-1),
inf));
    });
    old_xpos = ~x_pos;
}}
).play;
)
```

Figure 25.6
Use the mouse pointer to navigate the track along the *x*-axis of the screen.

We can then navigate our orderings by using the *x*-axis of the screen as the timeline of our original track. At each time point, the sound file is played back with the 1-s segments ordered by similarity.

To do this, we use `QtGUI` to track the position of the mouse pointer on the screen and then scale the position to the number of segments that we have, producing values between 0 and 221.

We make `Pbindef`, which plays back the sound file in 1-s chunks with a parameter for `startpos`. Our `Pseq` in `startpos` plays through the ordered data at the index of the mouse pointer. We update our `x_pos` and `Pbindef` only once every second, and when the mouse pointer moves to a new position. To do this, we save our `x_pos` on each loop so that we can check our new `x_pos` against the previous value.

We can hear certain clusterings of the sound file, with segments with similar vectors or pitch trajectories playing after each other. Each start position in the sound file produces a slightly different trajectory through the sound file.

25.3.3 Sound Classification

A common task for machine learning is to train models that can classify things into categories. In SuperCollider we can, for example, train a model to classify between different sounds, or between different gestures. Classification is a useful way to create interactive music systems that can observe real-time data and react to different inputs in varying ways. For example, we could create an instrument that listens to a drummer and adds different effects when it hears particular drums, sounds, or phrases.

For this and the next example, we will use the Fluid Manipulation Toolkit. You can download and install it from https://www.flucoma.org/ (Tremblay et al. 2022)

Here, we will look at an example that lets us train a model to classify sound where the model will respond to whatever instantaneous sound is currently being captured by the microphone, but doesn't respond to a more complicated understanding of how sounds change over time. It uses a model called a Multilayer Perceptron, a standard type of neural network inspired by the neurons that make up human and animal brains.

Now, the code will be explained one section at a time. (See figure 25.7.)

```
(
//[0] data management
s.waitForBoot({
   ~analysisBuffer = Buffer.alloc(s,13); // by default FluidMFCC
 will return 13 coefficients
   ~outputBuffer = Buffer.alloc(s,1); // a 1 frame buffer for   rec
eiving the classification on the server
   ~inData = FluidDataSet(s);
   ~labels = FluidLabelSet(s);
});
)

(
//[1] a function to get MFCCs from audio input
~analyseSound = {
   var mic = SoundIn.ar([0]);
   FluidMFCC.kr(mic,startCoeff:1);
};
SynthDef(\analysis_synth,{
   var mfccs = ~analyseSound.();
   FluidKrToBuf.kr(mfccs,~analysisBuffer);
}).play;
)
```

Figure 25.7
Initialization code for sound classification.

The first task is to create some storage for data (figure 25.7). FluCoMa conveniently provides the `FluidDataSet` and `FluidLabelSet` classes to help manage data. Lastly, we define a function for extracting audio features from the microphone input. We use `FluidMFCC`, which extracts Mel Frequency Cepstral Coefficients (MFCCs). Very roughly, MFCCs are a list of numbers that describe the shape of the sound spectrum, reflecting the timbre of a sound.

This code (figure 25.8) collects training data. It collects audio features for 1 s and stores them in a data set. It also labels these audio features with the number defined in `~label`. You can run it as many times as needed, changing `~label` for different sounds. For example, you might collect the sound of humming for label 0, rattling keys for the label *"class A,"* and the ambient background sound of the room for the label *"class B."* You can use as many labels and collect as much data as you like for each label. It's best to start off simply, with just two or three labels.

```
(
//[2] define a function for adding 30 labels over the course of 1 s
econd
~add_labels = {
   arg label;
   ~inData.size({
        arg size;
        fork{
              30.do{
                   arg i;
                   var id = size + i;
                   id.postln;
                   ~inData.addPoint(id,~analysisBuffer);
                   ~labels.addLabel(id,label);
                   30.reciprocal.wait;
              };
        };
   });
};
)

//[3] add some labels for a first class (try whistling for example)
~add_labels.("class A");

//[4] add some labels for a second class (try hissing "sssss" for e
xample)
~add_labels.("class B");
```

```
//[5] peek in on the data
(
~inData.print;
~labels.print;
)
```

Figure 25.8
Collection of training data.

The next task is to train the network (figure 25.9). First, we do some data conversion, and then we run `~nn.fit`, which trains the network, essentially trying to find the relationship between the source data and the labels. When you train the network, the training error will be shown in the post window. You can keep training until this value is low enough (< 0.05 is good in this case). If you can't get the error to go this low, then perhaps there are some problems with the training data. For example, you may have two different labels for two similar sounds. If this is the case, then just start again and collect new data.

```
//[6] initialise a neural network
(
~nn = FluidMLPClassifier(s,[8],activation: FluidMLPClassifier.   ta
nh, maxIter: 1000,learnRate: 0.1,momentum: 0.9,batchSize:   8,valid
ation: 0);
)

(
//[7] keep running this until the error is small (< 0.05)
~nn.fit(~inData,~labels,{|x| [\error,x].postln;});
)
```

Figure 25.9
Training the neural network.

In this final code (figure 25.10), we use our trained network. This synth listens to the microphone, predicts the label of the current audio input, and then uses the label to control a sound. In this case, it controls the cutoff of a low-pass filter.

```
//[8] run this synth to classify the incoming sound and adjust the
 filter according to the classification
(
{
   var oscs, mfccs, trig = Impulse.kr(SampleRate.ir/512),   classif
ication;
```

```
    mfccs = ~analyseSound.();
    FluidKrToBuf.kr(mfccs,~analysisBuffer);
    ~nn.kr(trig, ~analysisBuffer, ~outputBuffer);
    classification = FluidBufToKr.kr(~outputBuffer);
    classification.poll; // 0 is the "zeroth" label introduced   to
 the LabelSet, 1 is the "first" label introduced, etc.
    oscs = ({Saw.ar(100 + 5.0.rand)}!20).mean!2;
    BMoog.ar(oscs, classification.lag2.madd(2000,500), 0.2)
}.play;
)
```

Figure 25.10
Classifying live sound with the trained neural network.

Reader exercises:

• Experiment with different audio features instead of MFCCs You can tune the training data so the neural network responds to different aspects of sound. For example, `FluidChroma` gives data about pitch, or `FluidLoudness` responds to amplitude.

• Map the labels to different synthesis parameters. Tune the code in figure 25.11 so the results of classification have different effects (e.g., triggering samples or changing effects).

25.3.4 Mapping Movement to Sound

This next example continues with the Fluid Manipulation framework and looks at training a neural network with continuously varying data, also known as *regression*. This process is similar to training a classifier, except that rather than having discrete labels as target outputs for the neural network, we have continuous and varying streams of data, possibly in multiple dimensions. The example that we will focus on here maps motion into sound. In this case, the motion comes from your hand with your computer's mouse or trackpad, a two-dimensional stream of numbers (X and Y coordinates). The output from the network will be parameters for a synthesiser that uses Phase Modulation (PM) synthesis, controlled by 9 parameters. PM synthesis and its cousin FM synthesis, are well known for being somewhat unintuitive to control; i.e., the parameter spaces can be quite nonlinear, and it's not always clear as to which combination of settings will achieve a particular sound. ML can be really helpful here to let us explore PM synthesis in a more intuitive way; we will use a neural network to map gestures to sounds and then explore the space between the sounds with gestures. We will create a training set that pairs up different areas on the screen (with different mouse coordinates) with different sound settings on the PM synth, and then train a neural network that estimates the relationship between motion and sound. The training set need not cover all the areas

on the screen, as the trained network will provide estimates for unseen data, a bit like interpolating between patches. Sometimes these trained mappings will contain interesting and surprising sounds.

Next, we'll look at the details of the code, section by section.

Figure 25.11 defines a PM synth. All the parameters are normalized to between 0 and 1.0, which makes it easier to interface with a neural network. The synth creates a stack of three `PMOsc` UGens, each one being used as an input for the next. This synth has a very wide variety of sounds, with a large 9-dimensional parameter space to explore. The code also initializes the objects we need for data collection. The last piece of code runs the synth.

```
(
//[0] data management
~inputBuffer = Buffer.alloc(s,2);
~outputBuffer = Buffer.loadCollection(s,[0.1, 0.9, 0.2, 0.3,   0.8,
0.3, 0.2, 0.7, 0.2]); // 9 control parameters
~inData = FluidDataSet(s);
~outData = FluidDataSet(s);
)

(
//[1] make a complex synth with multiple (unintuitive) params
~pmsynth = {
   var p = FluidBufToKr.kr(~outputBuffer);
   var op1 = PMOsc.ar(p[0].linexp(0,1,20,2000), p[1].linexp(0,1,
 0.1,2000), p[2]*pi*2,0);
   var op2 = PMOsc.ar(p[3].linexp(0,1,0.1,200),p[4].linexp(0,1,
 0.1,2000), p[5]*pi*2, op1);
   PMOsc.ar(p[6].linexp(0,1,0.1,200),p[7].linexp(0,1,0.1,2000),   p
[8]*pi*2, op2);
};
SynthDef(\pmsynth, {
   Out.ar(0,~pmsynth.()!2);
}).play;
)
```

Figure 25.11
Code for initializing the neural network and defining a phase modulation synthesizer.

Now, we collect training data. The code in step 2 randomises the synth settings, which is a way of exploring the different sounds in the synth. When we find a sound we like, we can collect data that pairs this set up parameters with mouse coordinates. To do

this, we run the code in step [3], which collects a pair of mouse coordinates and stores this data along with the synth parameters. You can continue to run steps [2] and [3], to pair mouse coordinates and sounds. For example, you can associate one sound with the top left corner of the screen and another with the bottom right. It's best to start in a simple way like this, and slowly expand the complexity of the mappings. (See figure 25.12.)

```
(
//[2] explore the parameter space randomly. Choose a sound that you
like
~outputBuffer.setn(0, {1.0.rand}! 9);
)

(
//[3] This synth will collect mouse coordinates to associate with t
he current sound
~inData.size({|dataSize|
   {
        FluidKrToBuf.kr([MouseX.kr, MouseY.kr],~inputBuffer);
        FluidDataSetWr.kr(~inData, idNumber: dataSize, buf:   ~inpu
tBuffer, trig:1);
        Line.kr(dur:ControlDur.ir, doneAction:2);
   }.play;
        ~outData.addPoint(dataSize,~outputBuffer);
});
)

//[4] repeat steps [2] and [3] to collect data that associates mous
e positions with sounds

(
// (peek in on the datasets if you want)
~inData.print;
~outData.print;
)
```

Figure 25.12
Collection of training data.

The next step is to stop the synth, and then train the neural network. The training error will be shown in the post window. The network needs to be trained until the error is very low. Bear in mind that, in this exercise, there's no perfect model. Part of the value of using a neural network here is in the estimates that we receive for unseen data. A

partly trained network may sound very good and have some interesting surprises, so it's good to experiment with different levels of training. (See figure 25.13.)

```
(
//[5] initialise a neural network
~nn = FluidMLPRegressor(s,[7],activation: 1,outputActivation:   1,m
axIter: 1000,learnRate: 0.1,momentum: 0,batchSize:   2,validation:
 0);
)

(
//[6] train: keep running this until the error is small (< 0.05)
~nn.fit(~inData,~outData,{|x| [\error,x].postln;});
)
```

Figure 25.13
Training the neural network.

Lastly, we define a synth that controls sound based on predictions from the trained neural network. If the network has trained well, you should hear something close to the original mouse coordinate-sound pairs from the training set, with the addition of interpolations between those sounds for the screen areas that were not present in the original data. We have used ML to create a gestural instrument. (See figure 25.14.)

```
(
//[7] stop the current synth, and run this one instead
//this synth predicts the synth parameters based on the mouse posit
ion, using the neural network
SynthDef(\pmsynthPrediction, {
   FluidKrToBuf.kr([MouseX.kr, MouseY.kr],~inputBuffer);
   ~nn.kr(Impulse.kr(30), ~inputBuffer, ~outputBuffer);
   Out.ar(0,~pmsynth.()! 2);
}).play;
)
```

Figure 25.14
Using the trained model for mapping mouse movements into synthesis parameters.

Here are some ideas for further exploration of this example:

- Try more complex training sets with more sounds associated with more screen areas. The more complex the training data, the harder it will be for the neural network to train, but it may lead to potentially more interesting results.

• Try different input devices (e.g., a joystick (using an `HID` object) or a MIDI controller).

• You can control a trained network with UGens instead of the mouse (e.g., with low-frequency oscillators or `Line.kr`).

• Try adding more `PMOsc` ugens into the synth for a more complex sound and larger parameter space. You could also try another synthesis method with a large parameter space (e.g., additive synthesis or spectral processing).

25.3.5 Interfacing with Models Outside of SuperCollider: An Example of Generative Deep Learning

In the world of machine learning, it's common to use tools and frameworks in languages such as Python, JavaScript, or C++ to train and run models. There are many established code libraries that we may want to use, as well as many pretrained models that are available to run. Some of these give access to the GPU, which is not available in SuperCollider and is helpful for fast parallel processing to run the large models that are often needed with contemporary Deep Learning techniques (Briot et al, 2020, Goodfellow et al. 2016).

In this last example, we will demonstrate how to interface a model created in another system with SuperCollider. This is an example of unsupervised learning using a generative model called a variational autoencoder (VAE) (Kingma and Welling, 2019). In short, a VAE can be used to analyze the underlying structure of a set of data, and then to generate new variations based on that data. At the core of a VAE is a *latent space*. This is a multidimensional vector of numbers, which provides some control over what the VAE generates.

VAEs can be used for a very wide range of creative applications, but in this example, we keep the VAE simple for demonstration purposes. The system runs within Keras, a machine learning framework, and is scripted in the Python 3 language. We provide a pretrained model, although you can train one yourself using the code provided in the Jupyter notebook accompanying this chapter. The data set is a set of spectral frames, resulting from the analysis of an audio file (in this case, the music box example from the Fluid Manipulation Framework). This VAE has a two dimensional latent space. After training, the VAE generates spectral frames based on the position in latent space. The python script sets up the VAE in a local server that receives commands with Open Sound Control. We can navigate this space using mouse coordinates, so the script in supercollider sends mouse x and y to the VAE, and it responds with a spectrum. In SuperCollider, this is used as a filter for white noise. The result is an instrument that navigates across sonic textures, based on the tonal qualities of the sound that the model was trained with. To run this example, run the python script from the command line:

```
python vae.py
```

The SuperCollider code works as follows (figure 25.15).

```
(
n = NetAddr("127.0.0.1", 57120); // local machine
~magbus = Bus.control(s, 512);
SynthDef(\vaesynth, {
   var in, chain;
   SendReply.kr(Impulse.kr(20), '/mouse', [MouseX.kr(-2,2),   Mouse
Y.kr(-2,2)]);
   in = WhiteNoise.ar(0.8);
   chain = FFT(LocalBuf(1024), in); // encode to frequency domain
   chain = chain.pvcalc(512, {|mags, phases|
        [~magbus.kr, phases].flop.clump(2).flop.flatten
   }, tobin: 512);
   Out.ar(0,IFFT(chain)!2); // decode to time domain
}).add;
)
```

Figure 25.15
A synth that captures mouse coordinates, sends them to the model, and receives sound spectra back from the model.

We begin by setting up a control bus, to map the spectral frames to a synth on the server, and then we define this synth. It plays white noise and then filters this with the received spectral data using chain.pvcalc. It also sends the mouse coordinates to the editor using SendReply.

The final step is to run the synth and then set up two OSC functions. Here, \mouseReceive takes mouse coordinates from the synth and forwards them to the VAE server, which then runs the model using these values as coordinates in the latent space. Then \vaeReceiver receives spectral data back from the VAE server and sends them to the synth to be used as a filter (figure 25.16).

```
(
~vaesynth = Synth(\vaesynth);
)

(

OSCdef(\vaeReceiver, {|msg, time, addr, recvport|
   msg[1..].postln;
```

```
    ~magbus.setn(msg[1..] * 4000); //scale up values
}, '/spec');

OSCdef(\mouseReceive, {|msg, time, addr, recvport|
    var mousex, mousey;
    var vaeServer = NetAddr("127.0.0.1", 57030);
    mousex = msg[3];
    mousey = msg[4];
    vaeServer.sendMsg("/vae", mousex, mousey);

}, '/mouse', s.addr);

)
```

Figure 25.16
OSC communication between SuperCollider and the VAE model.

25.4 Summary

Machine Learning is a huge and fast-moving field of research and practice. In this chapter, we've presented a lightning introduction to the foundations of ML and given practical examples that demonstrate basic ML in SuperCollider. One would expect many new techniques, architectures, and creative possibilities to arise in the future that will probably move far beyond what we've shown here. However, what we do hope to have demonstrated is the core process by which we can use ML techniques in SuperCollider. We also hope that through trying these examples, you will have experienced how ML can be used and experimented with as a creative material, and how, although sometimes it may be strange and unpredictable, it can yield serendipitous results in the creation of sound, music, and instruments.

References

Baalman, M., 2020. "The Machine Is Learning." In *Proceedings of the International Conference on Live Interfaces*, Trondheim, Norway.

Bengio, Y., I. Goodfellow, and A. Courville. 2017. *Deep Learning* (Vol. 1). Cambridge, MA: MIT Press.

Bernardo, F., M. Zbyszyński, M. Grierson, and R. Fiebrink. 2020. "Designing and Evaluating the Usability of a Machine Learning API for Rapid Prototyping Music Technology." *Frontiers in Artificial Intelligence,* 3: 13.

Briot, J. P., G. Hadjeres, and F. D. Pachet. 2020. *Deep Learning Techniques for Music Generation* (Vol. 1). Heidelberg, Germany: Springer.

Caillon, A., and P. Esling. 2021. "RAVE: A Variational Autoencoder for Fast and High-Quality Neural Audio Synthesis." *arXiv preprint arXiv:2111.05011*.

Collins, N. 2011. "SCMIR: A SuperCollider Music Information Retrieval Library." In *Proceedings of ICMC2011, International Computer Music Conference*, Huddersfield, UK.

Collins, N. 2016. “Towards Machine Musicians Who Have Listened to More Music than Us: Audio Database-Led Algorithmic Criticism for Automatic Composition and Live Concert Systems.” *Computers in Entertainment,* 14(3): 1–14.

Fiebrink, R. 2017. “Machine Learning as Meta-Instrument: Human-Machine Partnerships Shaping Expressive Instrumental Creation.” In T. Bovermann, A. de Campo, H. Egermann, S.-I. Hardjowirogo, and S. Weinzierl, eds. *Musical Instruments in the 21st Century* (pp. 137–151). Springer.

Fiebrink, R., and L. Sonami. 2020. “Reflections on Eight Years of Instrument Creation with Machine Learning.” In *NIME 2020*.

Fiebrink, R., D. Trueman, and P. R. Cook. 2009. “A Meta-Instrument for Interactive, On-the-Fly Machine Learning.” *Proceedings of the International Conference on New Interfaces for Musical Expression*.

Goertzel, B., and C. Pennachin. 2007. *Artificial General Intelligence*. New York: Springer.

Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. Cambridge, MA: MIT Press.

Hiller, L. A., and L. M. Isaacson. 1957. “Musical Composition with a High Speed Digital Computer.” In *Audio Engineering Society Convention,* 9.

Kiefer, C. 2012. “Multiparametric Interfaces for Fine-Grained Control of Digital Music.” Doctoral diss., University of Sussex, Brighton, UK.

Kingma, D. P., and M. Welling. 2019. “An Introduction to Variational Autoencoders.” *Foundations and Trends in Machine Learning,* 12(4): 307–392.

Knotts, S., and N. Collins. 2020. “A Survey on the Uptake of Music AI Software.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*.

Lévy, B., G. Bloch, and G. Assayag. 2012. “OMaxist Dialectics.” In *New Interfaces for Musical Expression*.

Lewis, G. E. 2000. “Too Many Notes: Computers, Complexity and Culture in Voyager.” *Leonardo Music Journal*, 10: 33–39.

McCallum, L., and M. S. Grierson. 2020. “Supporting Interactive Machine Learning Approaches to Building Musical Instruments in the Browser.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Birmingham, UK.

Miranda, E. R., ed. 2021. *Handbook of Artificial Intelligence for Music: Foundations, Advanced Approaches, and Developments for Creativity*. Springer Nature.

Pachet, F. 2003. “The Continuator: Musical Interaction with Style.” *Journal of New Music Research,* 32(3): 333–341.

Roberts, A., J. Engel, Y. Mann, et al. 2019. “Magenta Studio: Augmenting Creativity with Deep Learning in Ableton Live.” In *Proceedings of the International Workshop on Musical Metacreation.*

Russell, S., and P. Norvig. 2020. *Artificial Intelligence: A Modern Approach,* 4th ed. Pearson.

Spiegel, L. 1986. “Music Mouse™—An Intelligent Instrument.” http://retiary.org/ls/programs.html.

Tatar, K., and P. Pasquier. 2017. “MASOM: A Musical Agent Architecture Based on Self Organizing Maps, Affective Computing, and Variable Markov Models.” In *Proceedings of the 5th International Workshop on Musical Metacreation*, Atlanta.

Tatar, K., P. Pasquier, and R. Siu. 2018. “REVIVE: An Audio-Visual Performance with Musical and Visual AI Agents.” *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*.

Tremblay, P. A., G. Roma, and O. Green. 2021. “Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit.” *Computer Music Journal*, 45(2): 9–23.

Viglensoni, G., L. McCallum, E. Maestre, and R. Fiebrink. 2022. “Contemporary Music Genre Rhythm Generation with Machine Learning.” *Journal of Creative Music Systems* 1(1).

Xambó, A., G. Roma, S. Roig, and E. Solaz. 2021. “Live Coding with the Cloud and a Virtual Agent.” In *Proceedings of NIME 2021,* https://doi.org/10.21428/92fbeb44.64c9f217.

26 Notations and Score-Making

Tom Hall, Newton Armstrong, and Richard Hoadley

26.1 Introduction

In 1959 Hiller and Isaacson speculated on the future application of computers to music, noting that "[the use of computer] by the composer to produce a written score as well as recorded sound should be highly efficient compared with current methods of composition" (Hiller and Isaacson, 1959, p.175). Aside from a few significant exceptions (see Weibel, 2020), the predominant focus of notational software development over the following decades was on fixed "traditional" common-practice Western notation (CPWN) (Mathews & Rosler, 1968; Roads, 1981).

Meanwhile, last century compositional and performance practice in the analog domain embraced notions both of so-called indeterminacy (see Behrman, 1965) as well as the live or dynamic score. For example, Earle Brown, reflecting many years later on his early iconic indeterminate composition, *December 1952*, explained how the composition's original conception was a motorized dynamic score in a box (Brown, 2008, p.3). Similarly, by 1969 the German composer Dieter Schnebel noted the projection of scores using slides in performance, whilst altering slide durations and appearance (Schnebel, 1969, p.291). It has only been in the twenty-first century that the notion of an algorithmic, interactive, dynamic or live computer-based "digital score" that may be used in similar performance contexts has also become well-established (see for e.g., Vear, 2019).

Our approach in this chapter is to consider the varieties of notational experience as–to paraphrase Iverson–tools of compositional thought (Iverson, 1980). In a sense both SuperCollider code and music notational outputs can be thought of as notations, however the focus here is on case studies which illustrate the use of SuperCollider to create both CPWN as well as examples of so-called "graphic notation," for which there has been renewed interest in recent years (see, for e.g., Sauer, 2009).[1] In doing so, we take in both "fixed" notation and scores, and live or dynamic interactive notational practices. Implicit in our approach is the hands-on interactive code-based experience afforded by the use of SuperCollider in the creation of a range of music notations. In general, we aim to highlight ways of making musical notation and scores that go beyond

modes of interaction using conventional music software, or present forms of notation that could not be easily attainable using such tools, e.g. dedicated CPWN software or DAWs.

We aim to demonstrate the use of SuperCollider's rich algorithmic paradigms within two primary contexts. The first is the use of SuperCollider alone to create different forms of musical notation outlined above; we begin with by considering approaches to using traditional music glyphs within a SuperCollider digital score "page," tackling Unicode and SMuFL fonts along the way. Changing tack, we then present a case study of an interactive project to make grid-based graphic scores within SuperCollider.

The two sections that follow present projects that use SuperCollider as an interactive environment to create notation and scores which are rendered visually in third party software. Using these approaches we are able to interact with other software using approaches and code interfaces native to SuperCollider, while also providing a straightforward interface for "plugging in" existing capabilities afforded through other software. A custom SuperCollider class that interacts with the INScore software is presented as a means to code audio and generative processes, allowing tightly synchronized synthesis and live scores. The `Fosc` project, on the other hand, enables complex and flexible compositional interaction primarily for the production of fixed LilyPond scores—whether algorithmically generated or "traditionally" composed.

26.2 CPWN and Graphic Notation within SuperCollider

26.2.1 Introduction

A benefit to working with music notations solely within SuperCollider is the tight integration between sound and image that can be obtained using the coding environment of a single piece of software. This section is divided into three parts: in the first we consider some low-level tools for managing SuperCollider as a "blank canvas" for score and notation creation. The second part introduces classes for working directly with CPWN within SuperCollider, and the last section presents a case study in graphic score notation.[2] To assist in creating responsive and structured compositional environments, the overall approach has been to use model–view–controller (MVC) and other related design patterns.[3]

26.2.2 Page, frame, view

For many, the blank screen now replaces the former blank page or canvas. In two-dimensional digital or printed work, the aspect ratio of the dimensions of a computer screen is a useful starting point, creating as they do the look and "frame" of a work's presentation. The SuperCollider helper class `MITHScreenRatios` assists in translation between the "frame" of pixel dimensions at a given ratio (e.g., 16:9) and a required

ratio and available size to fit within it. For example, the following code returns the dimensions of the largest frame with the ratio of an ANSI standard US Letter (11: 8.5) that will fit within the current screen pixel dimensions in landscape orientation: `MITHScreenRatios().ratioToDims(\usLetter)`. Below, the same ratio is used to set dimensions for a `Window` in the view class `MITHRatiosView`, then resized to a portrait-oriented *iso* ratio (`[sqrt(2): 1]`, e.g. A4) frame, adjusted to fit below the OS menu bar (on macOS and similar):

```
m = MITHScreenRatios(); m.viewDims_(m.ratioToDims(\usLetter));   MI
THRatiosView(m);
m.viewDims_(m.ratioToDims(\iso, landscape: true));
```

Whilst `MITHScreenRatios` assists with set-up of a digital page, `ViewCentral` and the MVC "façade" design (Gamma et al., 1995) subclass `WinBlock` are tools for presenting work within a digital page centered within a full-screened window. (This is achieved by nesting `VLayout` and `HLayout` in the view class.) The "block" models the traditional textblock used in page design for text or music, i.e. the area surrounded by the margins (see Bringhurst, 1999, chapter 8). As suggested in figure 26.1, using the class' `gui` method, the block can be moved within the page to create an intentional design. We return to these classes in the next subsections.

```
// dimensions for an iso ratio landscape Window, width 800px
~dims = MITHScreenRatios().ratioToDims(\iso, 800) // default is
 m.screenDims

// make a WinBlock, args: win, dims, margins, usrViewBool
b = WinBlock.new(dims: ~dims); b.moveWin(0, 0)

// view is by default an UserView
(
b.marginCol_(Color.white);
b.view.drawFunc = {|self|
   Pen.addRect(Rect(0.5, 0.5, self.bounds.width-1, self.bounds.   h
eight-1));
   Pen.stroke;
};
b.view.refresh;
)

b.viewDims // return current block dimensions

// experiment with blockSize / ratio
```

```
g = b.gui

// shape view dimensions
d = MITHScreenRatios().ratioToDims(\r5_3, ~dims * (5/6));
b.viewDims_(*d); b.centerView;

// show area revealed in fullscreen mode
b.winCol_(Color.new255(255, 85, 0)); b.resizeWin(*(~dims * 1.2));

// timed fullScreen, also ESC key (MBP) to endFullScreen
r {b.fullScreen; 3.wait; b.endFullScreen}.play(AppClock)

b.shrinkWin // hide area beyond main margin
g.close; b.close;
```

Figure 26.1
`WinBlock` used with `MITHScreenRatios`.

26.2.3 Unicode and common-practice Western notations

Having introduced simple tools for shaping and presenting the digital page, we next consider tools to assist working in SuperCollider with extended character sets such as those required for displaying CPWN music notation fonts. The easiest way to display CPWN musical symbols–"glyphs"–within SuperCollider is in a `Window`, within which a specialist music font can be specified for displayed `Strings`.[4] Since these characters are non-ASCII, it is irksome to enter them via a computer keyboard. The Unicode standard assigns a hexadecimal "code point" to individual symbols and character sets.[5] The class `MITHUnicode` assists with translation between Unicode code points and their encoding within SC as multi-byte UTF-8 characters (see also the related Quark `Strang` which shares some functionality). As with `ControlSpec`, the main way of accessing the class is via methods to other classes, e.g `String`. For the "musical keyboard" Unicode character: "🎹".`asCodePoint // U+1F3B9 and "U+1F3B9".asGlyph //` 🎹.

Standard Music Font Layout fonts ("SMuFL")[6] standardize access to music notation glyphs via Unicode code points. An example is Steinberg's "Bravura" font.[7] Once the font is downloaded and installed on a computer, the code below should display a treble clef[8] in a SuperCollider `Window`; also shown is concatenation of glyphs and ASCII text.

```
(
w = Window.new.front;
a = StaticText(w, Rect(100, 50, 300, 300));
a.font = Font("Bravura", 72);
a.string = "U+E050".asGlyph
```

```
)
a.string = ["U+1F1FA", "U+1F1E6"].scramble.asUnicodeString
```

Since the number of glyphs is very large, the `SMuFLtools` classes enable convenient browsing and viewing of the SMuFL font Bravura. Font data is parsed from a downloadable JSON file[9] (a demo subset is included with the class) and font information and the glyph is rendered in a SuperCollider `Window`.

The façade class `MITHNoteViewer` uses `MITHUnicode` and `ViewCentral` to display CWMN dynamically using a SMuFL font (default is Bravura: install before using the class). There's also a default glyph mapping class `MITHNoteMidiMap`, which enables regular 12-TET notational display in a manner similar to Max software's nslider object.[10] (A custom mapping class could be substituted to display glyphs for, e.g. a SMuFL microtonal notation.) The simplest view shows a single note, added as a MIDI integer. Adding a new note clears the existing one, as does the applying the "clear" method. A Boolean flag is used to force display of accidentals; the default mapping uses sharp signs for accidentals, a negative MIDI integer spells the note as a flat. (This mapping can be overridden as desired by altering the `accidentalsMap` Event method variable, or through using the `addCustom` glyphs method.) Figure 26.2 code creates a view with eight columns, each of which is accessed by a slot integer, or else using the `addAll` method via an `Array`. The figure demonstrates note creation, removal and color being changed dynamically, as also shown in screenshot figure 26.3.

```
(
s.waitForBoot({
   SynthDef(\percSine, {|midi = 69, mul = 0.2, out=0|
        var sig, env = Env.perc(releaseTime:0.4);
        sig = SinOsc.ar(midi.midicps, mul: mul)
         * EnvGen.kr(env, doneAction: Done.freeSelf);
        Out.ar(out, [sig, sig])
   }).add;
   l = MITHNoteViewer.new(8, 350);
   a = [60, 62, -63, 65, 67, 69, 71, 72, -70, -68, 67, 65,   -63, 6
2, 60];
   r = Routine{
        l.fullScreen;
        1.wait;
        a.size.do{|i|
              Synth(\percSine, [\midi, a[i].abs]);
              l.add(a[i], slot: i.fold(0, 7), color:Color.red);
              if(p.notNil){l.glyphsCol_(Color.black, p)};
              p = i.fold(0, 7);
              (60/94).wait
```

```
            };
            l.glyphsCol_(Color.black, p);
            p = nil;
            l.endFullScreen;
            2.wait;
            l.clearAll;
            l.close
        };
        AppClock.play(r)
    })
    )
```

Figure 26.2
`MITHNoteViewer` dynamic note array presentation with `Synth`.

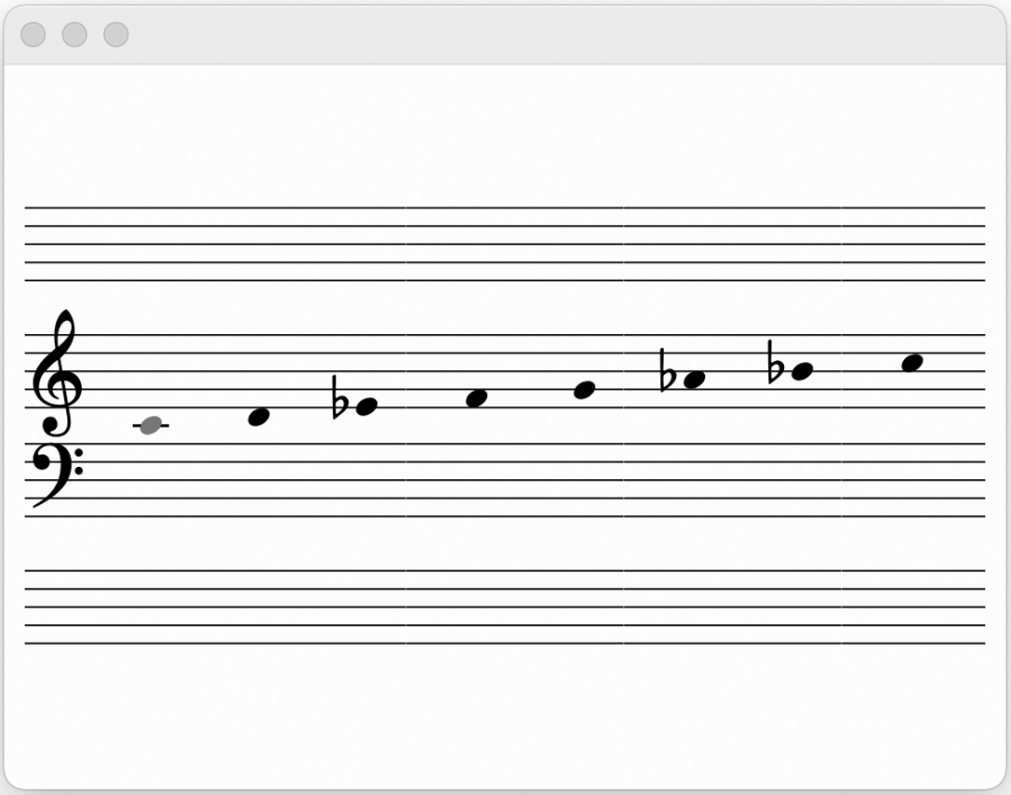


Figure 26.3
Screenshot of `MITHNoteViewer` as shown in figure 26.2.

26.2.4 Model, grid, score

Working directly with music glyphs purely within the SC environment soon becomes complex; possibilities for other music notations involving graphical symbols directly

created in SuperCollider are more programmatically manageable. The idea of a grid as a notational model for music goes back many centuries (Higgins, 2009), and can be thought of as a level of abstraction away from modern CWMN. A literal visual grid or graph was utilized by US composer Morton Feldman from 1950 in the score notation of his so-called graph music (Cline, 2016). A related graph approach later formed the basis of the model for the interactive GRIN (GRaphical INput) language cited in the chapter introduction (Mathews & Rosler, 1968). This section concludes with a project, `GMITH`, which uses SuperCollider to create a dynamic compositional environment for making Feldman-like notation and finished musical score.[11] The code design uses a hierarchy of MVC classes to draw a graphic score using the `Pen` class, and an `Array`-centric score class to represent the customizable graphical elements.

`GMITH`'s score class makes a distinction between the type of musical event presented within a grid and the customizable form of its presentation. Feldman's scores are similar to "piano roll" layout of musical events, in which an X-Y grid specifies units of time and pitch, respectively, except that the conception of pitch is somewhat indeterminate: precise pitch is not specified beyond high, medium or low (for any given instrument).[12] Feldman used three rows to indicate these relative registers, with integers within cells to indicate number of notes and various symbols or text to indicate playing technique (e.g., a diamond for a note played as a harmonic). Figure 26.4 shows a short example of the system, in which a performance event is represented as an `Association` of arrays for grid location and event type. Score events can also be added or removed singly via the add or remove methods (or clean slate via removeAll).

```
(
l = GMITH.new;
// [row, column] -> [number, type, duration]
l.addAll([
    [1, 0],
    [2, 1] -> [3],
    [0, 2] -> [nil, \h],
    [1, 3] -> ["Pz 3", nil, 2],
    [2, 5] -> [[2, 1]],
    [0, 6] -> [nil, \a]
]);
)

// |key, rows, cols, vHeightPct, win|
w = l.makeGalley(\gv1, cols: 8)
// to complete the score, click the bottom right (empty) cell, then
evaluate:
l.addMouseCell([nil, \p]);
```

```
// post sorted score events l.scoreArr
l.postEvents(order: 0) // row order. order: 1 posts column   order

// grid line patterns
l.gvGridPatterns(\gv1) // [[1], [1]]
l.gvGridPatterns_(\gv1, [1, 0, 0], [-1, 0, 0, 0]);

// display adjustments
l.gvColRange_(\gv1, 0, 63)
l.gvGridLineWidth(\gv1)
l.gvShapeLnWidth_(\gv1, 2) // adjust, default is 3
l.gvFontSizeScale_(\gv1, 0.8) // adjust
l.moveAll(12); // move score events to columns to the right
l.moveAll // return to 0 column as first event
l.gvColRange_(\gv1); // default view range is filled cell   range
```

Figure 26.4
Simple GMITH score creation and galley view display.

If the score is displayed, as here, with one or more "galley" views, clicking on a cell will post its `[row, column]` location and, if relevant, event information to the post window. Other interactive mouse actions which use modifier keys include <ALT>-click to remove a score event and <CTRL>-click to add a single cell event, or after clicking a cell location, use the method `addMouseCell` to create more complex or multi-cell events (see figure 26.4). Lastly, to move an event, the `move` method can be accessed using a mouse by clicking on an event, then <SHIFT>-clicking to a new cell location, which will move the last clicked event to this new location. Evaluating the `undo` method will revert to the previous score state for these actions.

Score events can be sorted by row or column and posted using the `postEvents` method. In order to create a flexible compositional environment, the galley view can be expanded, for example, to display empty cells using the `gvColRange_` method, and score events can be moved by column using the `moveAll` method. Morton Feldman varied the format of the grid display itself, but typically indicated groupings of 4 icti using dotted "bar" lines. The `gvGridPatterns` method allows customization of that part of the notation, in which arrays set the displayed pattern. Within these arrays, the integer 1 creates a solid line, 0 no line and -1 a dotted line. Solid lines (the default) is represented as `[1], [1]`. A variation of blank, dotted and solid lines is used in figure 26.4 (as seen also in the inner part of figure 26.5).

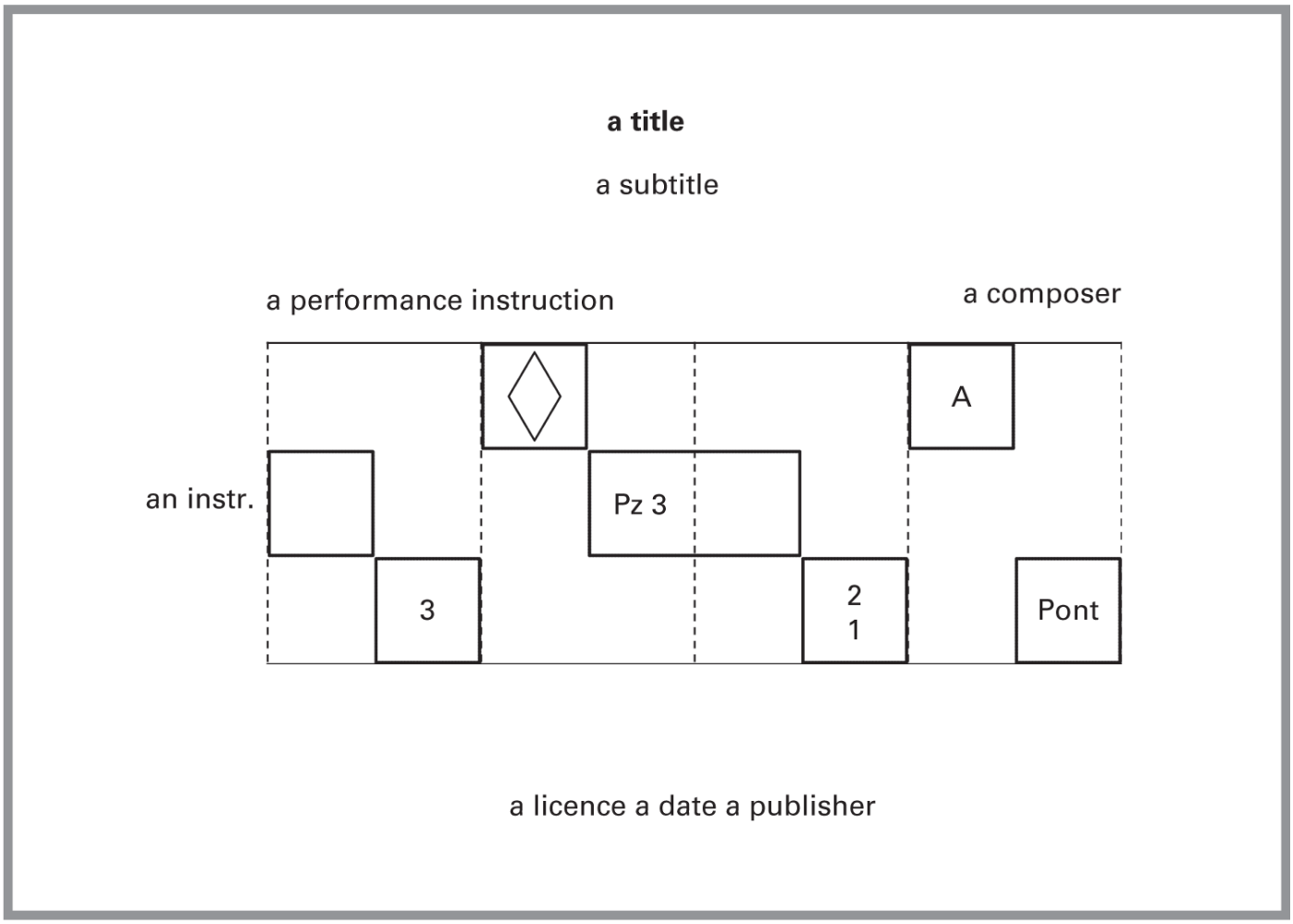


Figure 26.5
Example of GMITH score view using makePage method.

GMITH can also make a score comprising multiple screen-based "pages" for printing or dynamic display, as shown in figure 26.5 (code for this example is included in the chapter's code materials). As with the galley view, cell events are updated as relevant if the score is changed. This is achieved via a hierarchy of classes including WinBlock shown earlier, into which a custom class, NotnFlow is inserted. NotnFlow uses the Layout design as well as a class, TextModel as a model for placing StaticText into the page as needed. Page elements can be adjusted via the page's main model, as shown. An effective process using these classes is to decide view dimensions and margins, then adjust system(s) size (width and height) to match the size and ratio of the UserView that contains the notation system notation, as determined by pageSysSqSize_. Future work could include further consideration of row management and additional approaches to live dynamic display in a performance context.

26.3 INScore

26.3.1 Introduction

INScore is an "environment for the design of interactive, augmented, dynamic musical scores"[13] in which one can dynamically create, enhance and maneuver a variety of visual media, including graphic objects, text and musical CPWN. INScore's elasticity with these media, which are in the main created and manipulated programmatically, make it a useful notational tool.

A key motivation for using SuperCollider with INScore for musical composition is the ability to create unified interfaces enabling live generative audio processes to be synchronized in real-time with a live-generated performance score of arbitrary complexity (Hoadley, 2017). These processes are coded in SuperCollider using the custom classes, which communicate with INScore via the Open Sound Control (OSC) protocol (Wright & Freed, 1997).

We begin by demonstrating the creation of both CPWN and graphic images, with an emphasis on "dynamic" live notation. The concluding sections demonstrate the use of the software using a case study of a completed composition with links to online videos.

26.3.2 INScore and SuperCollider

Whilst there are a number of well-documented ways of working with INScore,[14] the approach taken here is to design composition and score data within SuperCollider, which is then sent to the INScore app for real-time display. Although INScore allows for two-way communication with another application using OSC,[15] in general it is more efficient to construct and transform INScore's object attributes within SuperCollider using the `INScoreSceneMIT` class.[16] This process can then be synchronized with audio generated in SuperCollider. For creating CPWN, we focus on the capabilities of INScore's embedded GUIDO engine,[17] which we demonstrate next.

Once the INScore app is downloaded and installed,[18] depending on your OS and installation location you should be able to open within SuperCollider by evaluating something like `"Applications/INScoreViewer.app".openOS`. The INScore app by default will open a Window scene called "scene" to which notational data can be sent. Working natively within the app, to display a middle-C quarter-note using GUIDO music notation ("gmn"), we could open a plain text file (extension "inscore") containing `/ITL/scene/score0 set gmn " [c]";`. To do this at a low level within SuperCollider using OSC, we set up a `NetAddr`, and send the message as follows:

```
// assumes INScoreViewer listening for OSC on port 7000
~addr = NetAddr.new("localhost", 7000);
~addr.sendMsg("/ITL/scene/score0", \set, \gmn, "[c]");
```

The `INScoreSceneMIT` class simplifies interaction with INScore and can also co-ordinate broadcasting to multiple computers. The following code uses this class to display the equivalent of the example above:

```
~scene = INScoreSceneMIT.new("localhost", 7000);
~scene.note(\c);
```

INScore allows multiple windows ("scenes"), each of which could be controlled by a `INScoreSceneMIT` class instance. Using the `addINScoreView` method allows synchronized control of a single scene across multiple computers, for example:

```
// synchronize with a remote machine
~scene.addINScoreView(NetAddr.new("192.168.178.64", 7000))   // nee
ds correct remote IP address
```

The class `INScoreManagerMIT` manages bi-directional communication with the INScore application, can be used to query the application on remote computers to confirm, for instance, scene window creation and deletion. This is useful when a remote screen is not visible, such as when separate computers are used for each performer, but managed by a primary computer. In cases, however, where only a local machine is being used, the `INScoreSceneMIT` class is adequate for basic scene creation and deletion. The following explicitly shows the second argument to the note method, `componentNum`, which in this case is used to create an INScore object `score0` to which the GUIDO note data string is sent, as follows:

```
j = INScoreSceneMIT.new("localhost", \joni).window
j.foreground;
j.note(pitch: "b&1/2 a", componentNum: 0);
j.close; // close window
```

Returning to our main window, the following shows a more complex example of a GUIDO string display using the `staves` method, where each voice is surrounded by square brackets:

```
// make chords manually (_ for rest), two staves
~scene= INScoreSceneMIT.new; // new class instance as needed
~scene.staves("[{a/4, b, c} {b3,e2, g&} _/8 {c#2/4., e0}],   [{g/1,
f}]");
```

This is followed by the use of a small GUIDO string manipulation helper class:

```
// helper string GUIDO string manipulation class
g = ~scene.guido; // otherwise: g = SimpleGuidoMIT.new;
~scene.note(pitch: g.clef(\f) + g.meter(3, 8) +g.key(-2) +   "c0/8.
b-1");
```

To create a "dynamic" score, once could create a function containing an algorithm to create a short unmetered sequence of chords. This is shown in figure 26.6, which uses the `midiNoteMap` method to translate between integers and GUIDO notation.[19]

```
~scene= INScoreSceneMIT.new; // new class instance as needed
(
~randomChordsFn = {
   var chordArr, chord3, noteFN, chosenDur, melodicArr;
   var guidoOutput = "", guidoChord = [], g = ~scene.guido;
   noteFN = {rrand(30, 90)};
   chordArr = [[50, 54, 57, 59], [34, 56, 78], 3.collect({noteFN.
value})];
   chord3 = [chordArr[1], chordArr[1]].choose;
   melodicArr = [chordArr[0], chordArr[0]+[2, -2].choose,   chord3,
chordArr[2]];
   melodicArr.do({|chord|
        chosenDur = ["/16", "/4", "/2."].choose;
        chord = chord.collect({|note| g.midiNoteMap(note,   "mixe
d") ++ chosenDur});
        chord = [g.makeChord(chord), "_"++chosenDur].   wchoose([0.
8, 0.2]);
        guidoOutput = guidoOutput + chord
   });
        guidoOutput;
};
~scene.osaActivate;
~scene.verbose_(false);
Routine.new({
   6.do({|i|
        ~scene.note(pitch: ~randomChordsFn.value);
        ~scene.htmlFull(fontsize: "40pt", text: (i+1) + "/ 6");
        ~scene.move(-0.4, -0.6, component:"html");
        2.5.wait
   });
   ~scene.clear; // clear existing scene
}).play;
)
```

Figure 26.6
Dynamic CPWN using a function to generate an algorithmic chord sequence.

Before we turn to other types of graphic objects, figure 26.7 presents a further CPWN example with audio, using a `Routine` to play a dynamic score in combination with a simple `Synth`.

```
~scene= INScoreSceneMIT.new // new class instance as needed
(
var acc, glyph, glyphRed, string, oldString;
s.waitForBoot({
   SynthDef(\percSine, {|midi = 69, mul = 0.2, out=0|
        var sig, env = Env.perc(releaseTime:0.4);
        sig = SinOsc.ar(midi.midicps, mul: mul)
        * EnvGen.kr(env, doneAction: Done.freeSelf);
        Out.ar(out, [sig, sig])
   }).add;
   m = [60, 62, -63, 65, 67, 69, 71, 72, -70, -68, 67, 65,   -63, 6
2, 60];
   g = ~scene.guido;
   string = g.stemsOff;
   oldString = string.copy;
   ~scene.osaActivate;
   r = Routine.new({
        1.wait;
        m.do({|i, j|
              Synth(\percSine, [\midi, i.abs]);
              acc = if(i.isNegative){"flats"}{""};
              glyph = g.midiNoteMap(i.abs, acc);
              glyphRed = g.noteFormat(color: \red) + glyph;
              // start a new string of pitches
              if(j==8){
                   string = g.stemsOff + g.midiNoteMap(m[j-1].abs,
acc);
                   oldString = string.copy;
              };
              string = (oldString + glyphRed).postln;
              oldString = oldString + glyph;
              ~scene.note(pitch: string);
              (60/94).wait
        });
        0.5.wait;
        ~scene.note(pitch: oldString);
        1.wait;
        ~scene.note(pitch: g.emptyStaff);
```

```
    }).play;
})
)
```

Figure 26.7
Dynamic score with audio.

INScore allows the creation and manipulation of a wide range of non-CPWN graphic objects (based on the open standard SVG XML-based image format),[20] regarding which the reader is directed to INScore's online documentation. The following examples demonstrate the creation and manipulation of a simple geometric shape. First we may wish to scale, move or delete the existing GUIDO `score0` scene object:

```
// scale, move or delete the existing GUIDO score0 scene object
~scene= INScoreSceneMIT.new // new class instance as needed
~scene.note(~scene.guido.emptyStaff); // from previous example
~scene.move(y: -0.8).scale(0.5);
~scene.deleteComponent(component: "score", componentNum: 0);
```

Next we can create a polygon and change its color as follows:[21]

```
~scene.poly([0.7, 0.05, 0.1, 0.1, 0.1, 0.0]);
~scene.color(\orange, component: "poly"); // https://en  .wikipedi
a.org/wiki/Web_colors
```

INScore's graphic space ranges by default from -1, to 1 in the usual dimensions, with 0 being the centre of a Window. An object's position can change by either relative or absolute positioning; the following places the centre of the polygon at the center or either horizontal edge of the scene window:

```
~scene.move(x: [-1, 0, 1].choose, component: "poly");
```

INScore's graphic space is designed to be stretchable: changing the Window size scales the size of the objects within it, and the graphic space can be extended or reduced from the default +/-1. This can cause complexities in scaling objects and windows on screen or via a projector in fullscreen mode. Building on the INScore *poly* object created above, figure 26.8 shows a screenshot from an example animation that combines multiple dynamic objects, including CPWN. The code is included in the chapter code materials on the book Web site. This example demonstrates a simple approach to screen ratio and window management; the example temporarily fullscreens the INScore window.

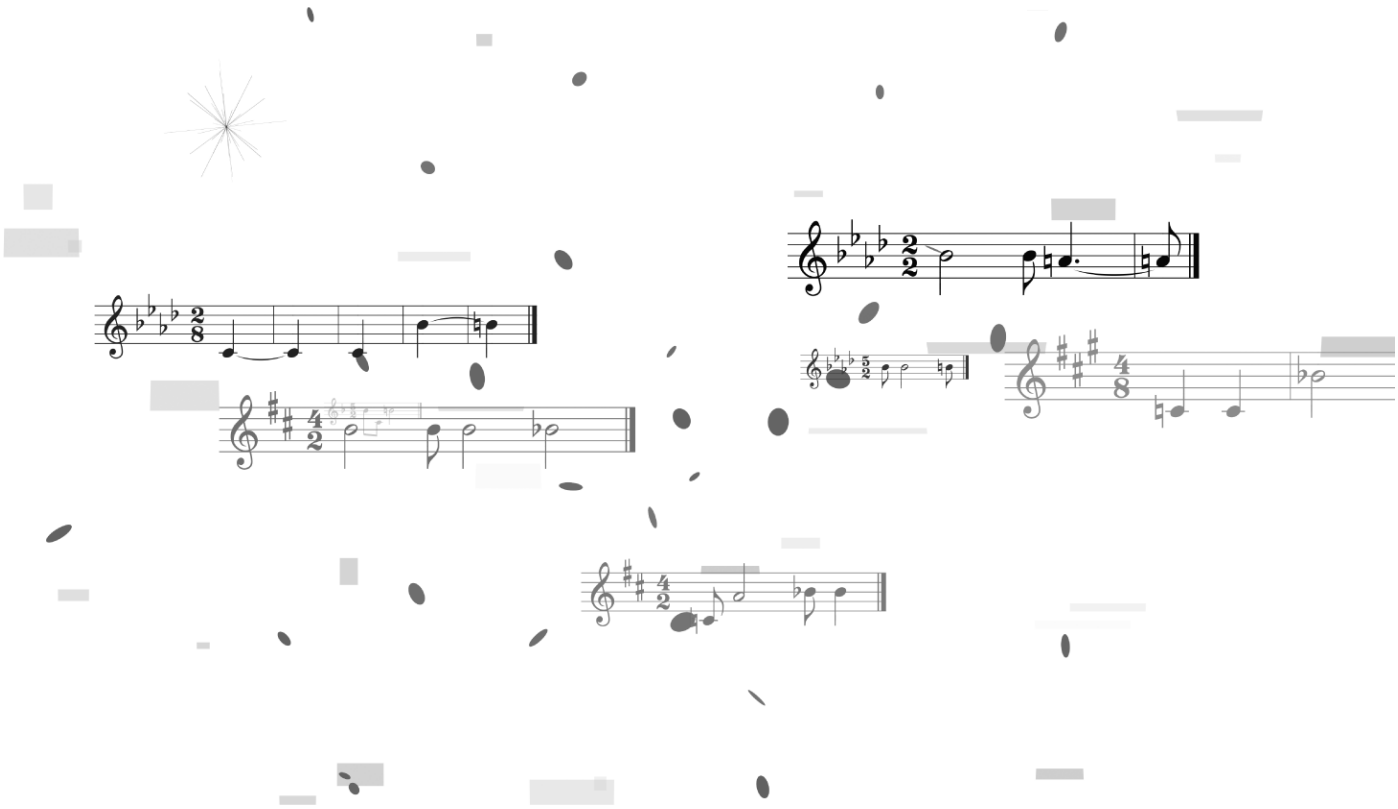



Figure 26.8
Screenshot for `INScore` scene scaling, object and dynamic CPWN animation.

26.3.3 Unthinking Things

The interactive nature of using INScore makes it a powerful tool for the creation and public performance of live, dynamic scores. In this scenario, musicians may either read from a separate computer screen or from a publicly displayed screen.[22] We next turn to a case study using SuperCollider and INScore in order to give a sense of an entire musical composition created using these tools.

Unthinking Things is written for sixteen-voice mixed choir, physical materials and electronics, one of a series of compositions by Hoadley involving dynamic, generative music notations.[23] The composition includes both CPWN, text and non-traditional graphic visual objects placed and moved within a "scene."[24] Displaying the influence of Cornelius Cardew's graphic score *The Great Learning* (Cardew: 1971), this approach allows non-metric music without requiring complex forms of CPWN. INScore allows the graceful and interconnected dynamic combination of all of these elements.

Figure 26.9 shows an example taken the "metals" section of the piece, which comprises a generative audio track made from looped recordings of singing bowls and the real-time sounds of the performers' striking resonant metal objects. The figure shows varied elongated isosceles triangles similar to those shown in section 26.3.2,

intentionally indicative of an amplitude envelope, but allowing interpretation by the performer according to the visual properties of the triangle.[25]

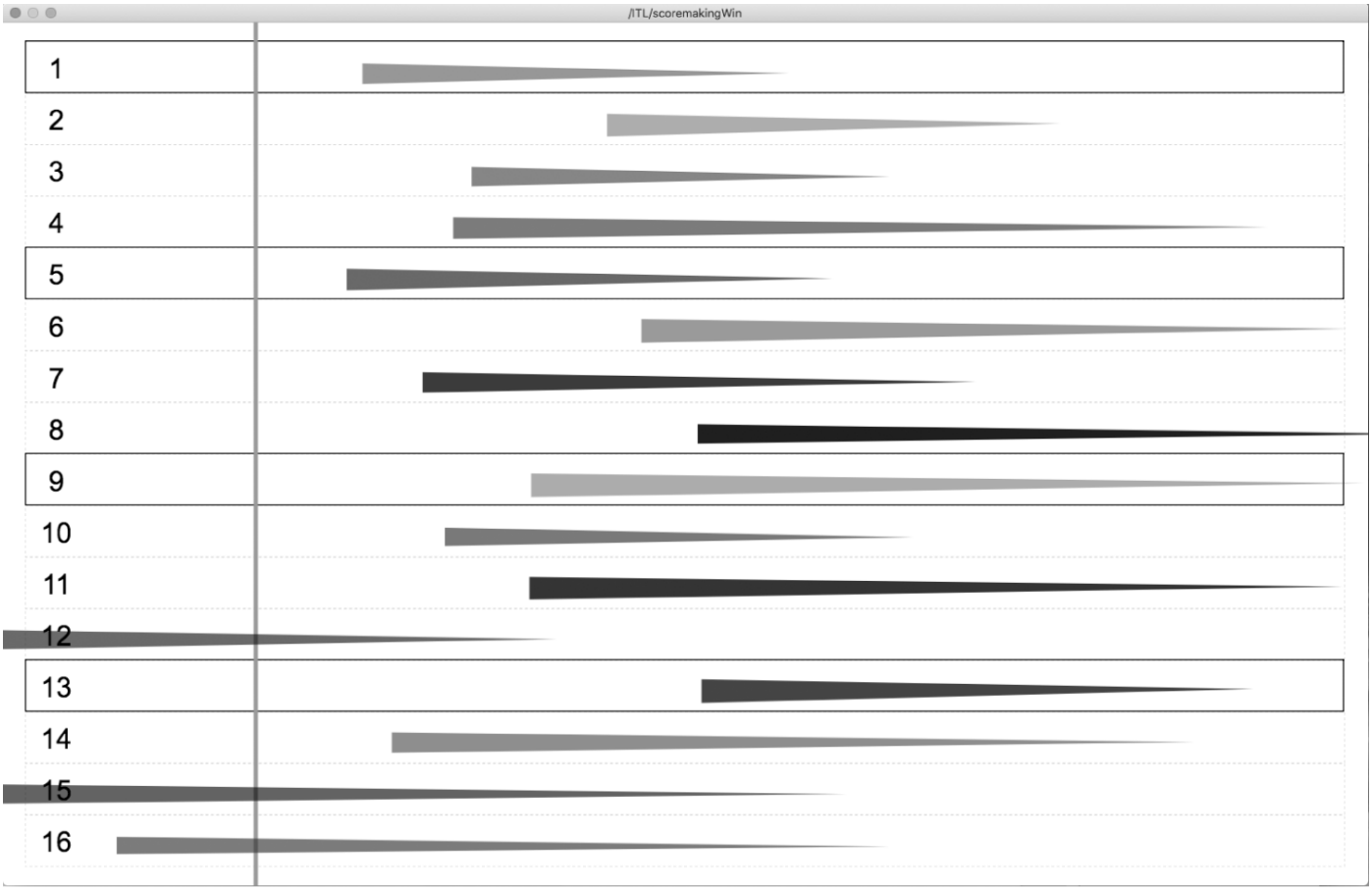


Figure 26.9
Screenshot taken from the "metals" section of *Unthinking Things*.

Given that the performer reads the notations in real time, there are two principal options for a dynamic score: the notations move against a static cursor, or a cursor moves across a field of static objects (the latter best reflecting standard fixed notation). The "metals" section in the piece uses the former approach, whereas the "stones" section uses the moving cursor model (figure 26.10). For this section, performers are asked to bring along stones which they can hit or rub together,[26] offering a sound palette of point-like, dry precision in contrast to the resonance of metals. As a result the algorithms of the "stones" section concentrate on rhythms, contrasting pulses and flock-like clouds of particles.

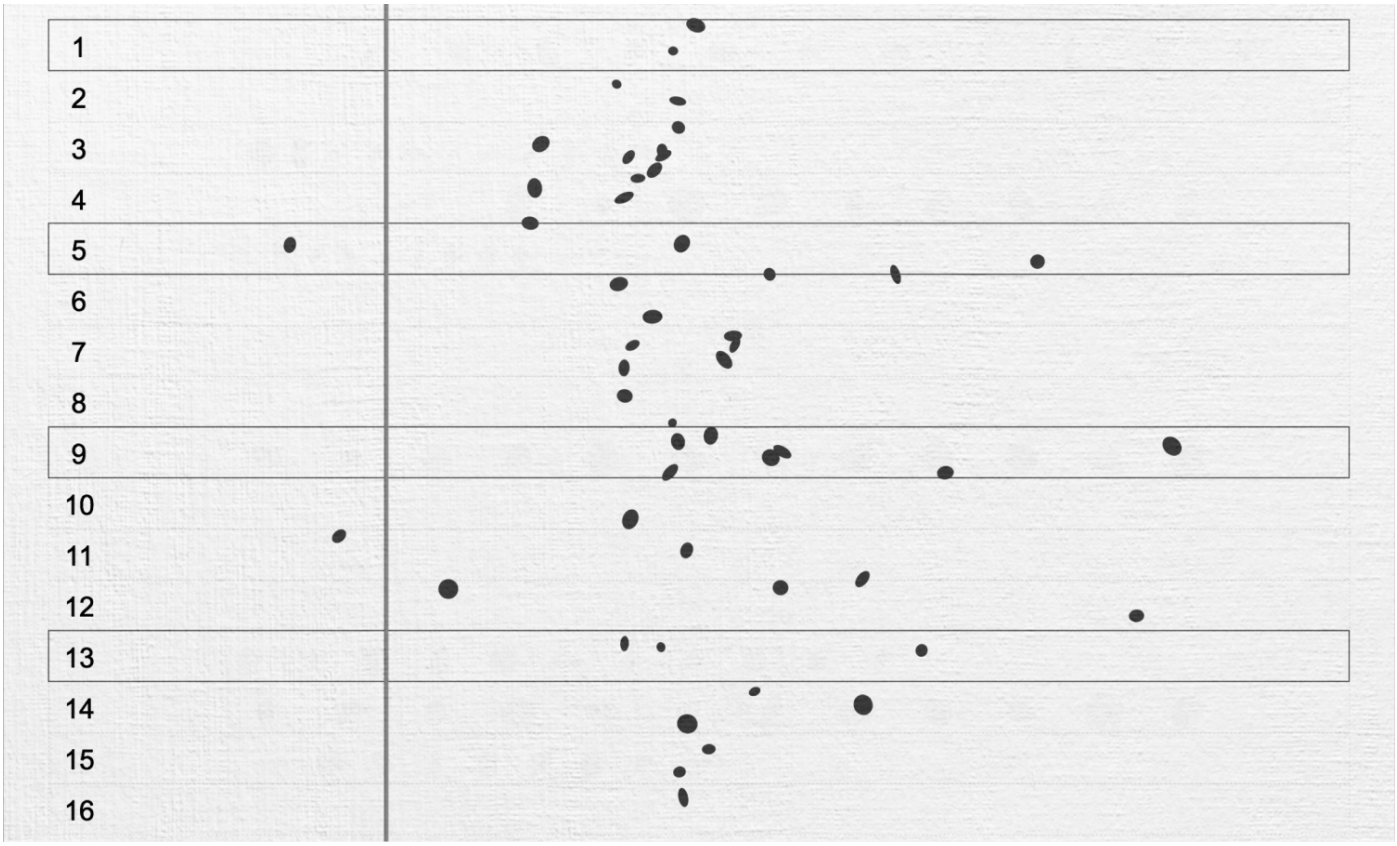
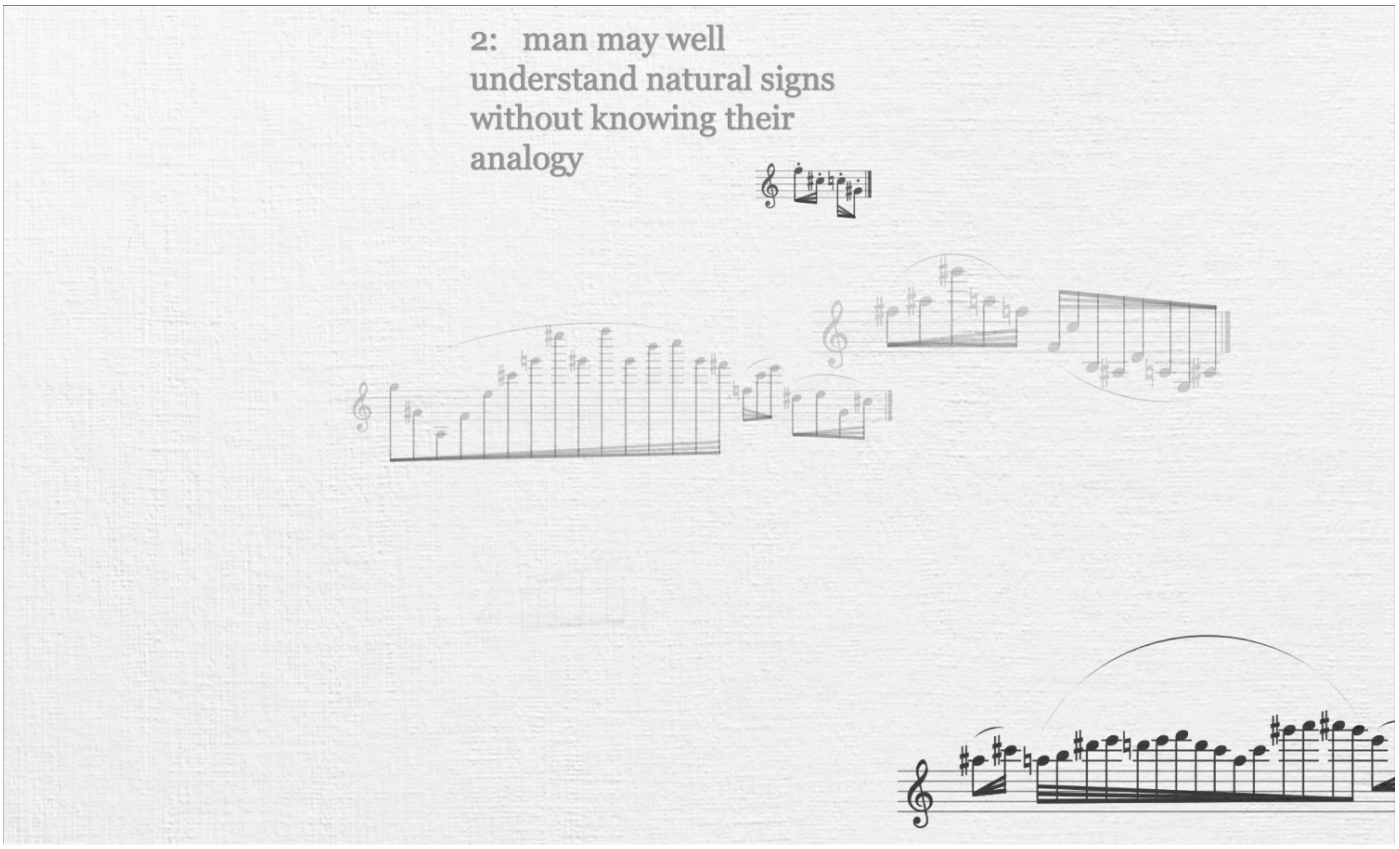


Figure 26.10
A stone "constellation" from *Unthinking Things*.

INScore is used in *Unthinking Things* to create a score with representative, decorative and performance-focused features. Figure 26.11 shows screenshots from sections using CWPN elements as in traditional performance scores. The first section (a) features a graphical intertwining of notations and text, the latter read live and so displayed in an eye-catching color prefixed by the performer's number. The music notation as presented is rendered by a variety of synthesized and sampled keyboard and harp-like sounds. Figure 26.11b shows an approach to multi-part writing for voices.



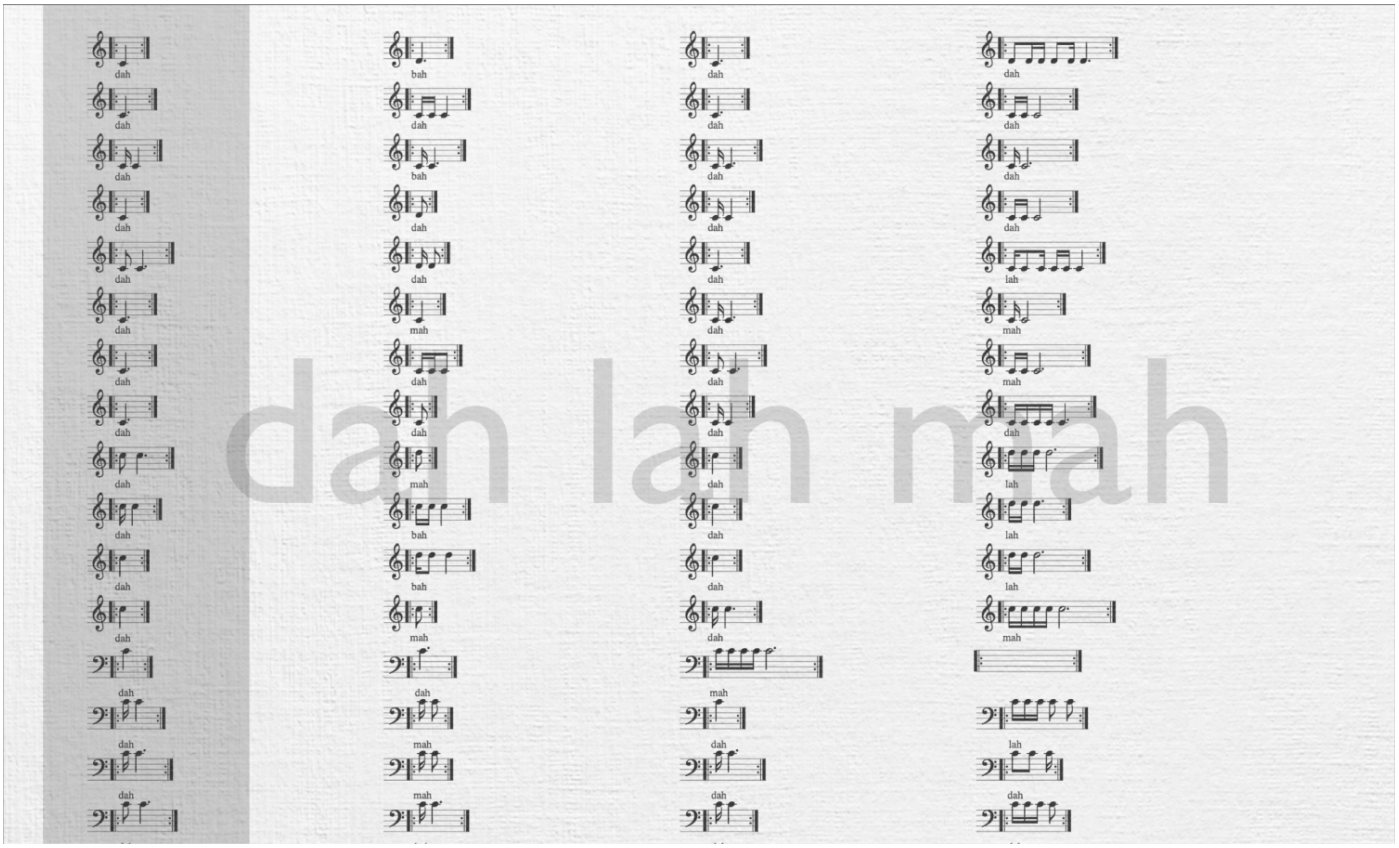


Figure 26.11
Other notations including CPWN from *Unthinking Things*. (a) text and CPWN ("melismata")[27]; (b) "vocal rhythms"[28].

26.4 Fosc

26.4.1 Introduction

Fosc (Formalised Score Control/For SuperCollider) is an Application Programming Interface for generating musical notation in LilyPond[29] using SuperCollider. Fosc is a close relative of Abjad[30] and ports much of Abjad's Python code base to SuperCollider. The motivation for the initial development of Fosc was to fill a gap in the software tools available within existing computer-assisted composition environments. In short, Fosc was designed to feel native to SuperCollider users.

Fosc provides an object-oriented interface to LilyPond's engraving library, allowing for the generation and transformation of musical scores of arbitrary complexity using extended CPWN. LilyPond's engraving library comprises a vast number of typographical elements. While Fosc provides low-level access to all of these elements, composers working with algorithmic systems will in most instances want to maintain focus on high-level concepts and constructs. To this end, Fosc provides a single, customisable and easily extensible factory class for the high-level algorithmic specification of musical materials.

Fosc has been used extensively for compositional projects since 2018 by Armstrong. One advantage enabled by the system is that it affords a wide range of styles of working. These range from "pure" approaches—in which the final score is generated entirely as a patchwork of algorithmically specified outputs (Armstrong, 2018)—to work (Armstrong, 2019) that emerges as a more reflective "back-and-forth" between formal specification and informal transcription and reinterpretation.

The section that follows begins with a short overview of low-level Fosc usage scenarios, and concludes with examples using the high-level FoscMusicMaker factory class. Fosc and LilyPond both need to be installed to run the code examples.[31]

26.4.2 Low-level interface

Fosc's low-level interface models musical notation at the level of notes, chords, voices, staves, staff groups, and the score. Figure 26.12 presents a simple example.

```
(
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8),   Fos
cNote(65, 1/2)];
FoscStaff(a).show;
)
```

Figure 26.12
Display notes on a staff.

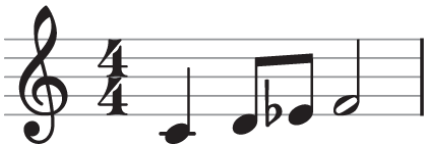



Figure 26.13
Result of code in figure 26.12.

Indicators of various types can be attached to the notes and chords in a musical score. Indicators include articulation and dynamic markings, along with many other standard musical typographical elements (time signatures, clefs, fermata signs, ottava markings, etc.). Figure 26.14 shows indicators attached to a `FoscNote`.

```
(
a = FoscNote(60, 1/4);
a.attach(FoscArticulation('>'));
a.attach(FoscDynamic('f'));
a.show;
)
```

Figure 26.14
Attach articulation and dynamics marks to a note.

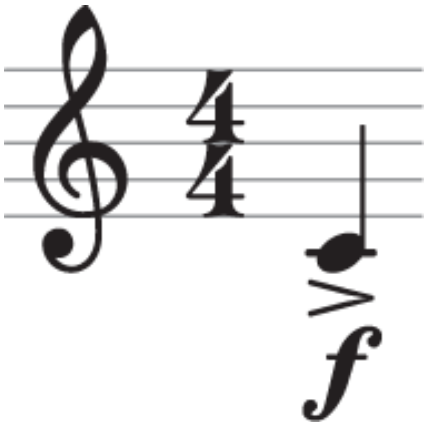



Figure 26.15
Result of code in figure 26.14.

Typographical elements that span sequences of notes and/or chords—such as slurs, ties, and dynamic hairpins—can be added using `Fosc`'s selection mechanism and spanner methods, as shown in figure 26.16.

```
(
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8),    Fos
```

```
cNote(65, 1/2)];
b = FoscStaff(a);
b.selectLeaves.slur.hairpin('p < f');
b.show;
)
```

Figure 26.16
Attach slur and dynamic hairpin spanners to a selection of notes.

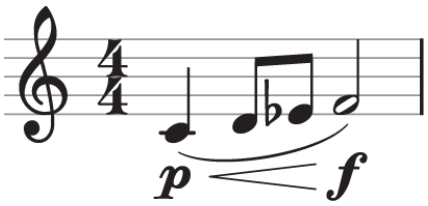



Figure 26.17
Result of code in figure 26.16.

Fine-grained formatting of typographical elements can be performed by attaching LilyPond literal strings directly to score objects (figure 26.18).

```
(
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8),   Fos
cNote(65, 1/2)];
b = FoscStaff(a);
b[0].attach(FoscLilyPondLiteral("\\tweak NoteHead.style   #'harmoni
c"));
b.show;
)
```

Figure 26.18
Attach a LilyPond literal string to the first note in a selection.

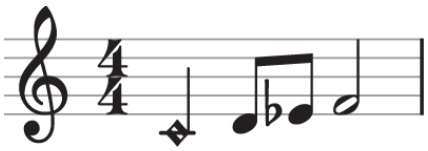



Figure 26.19
Result of code in figure 26.18.

`Fosc` supports a range of *mutation* methods for transforming selections of music in place. These include methods for pitch transposition and duration scaling, as shown in

figure 26.20, each part of which shows the original staff and mutated copy.

```
// Make a copy of a staff. Transpose the contents of the copied sta
ff up two semitones
(
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8),   Fos
cNote(65, 1/2)];
b = FoscStaff(a);
b.selectLeaves.slur.hairpin('p < f');
c = b.deepCopy;
mutate(c.selectLeaves).transpose(2);
FoscScore([b, c]).show;
)
// Make a copy of a staff. Scale the contents of the copied   staff
by a factor of 2
(
a = [FoscNote(60, 1/4), FoscNote(62, 1/8), FoscNote(63, 1/8),   Fos
cNote(65, 1/2)];
b = FoscStaff(a);
b.selectLeaves.slur.hairpin('p < f');
c = b.deepCopy;
mutate(c.selectLeaves).scale(2);
FoscScore([b, c]).show;
)
```

Figure 26.20
The `transpose` and `scale` mutation methods.

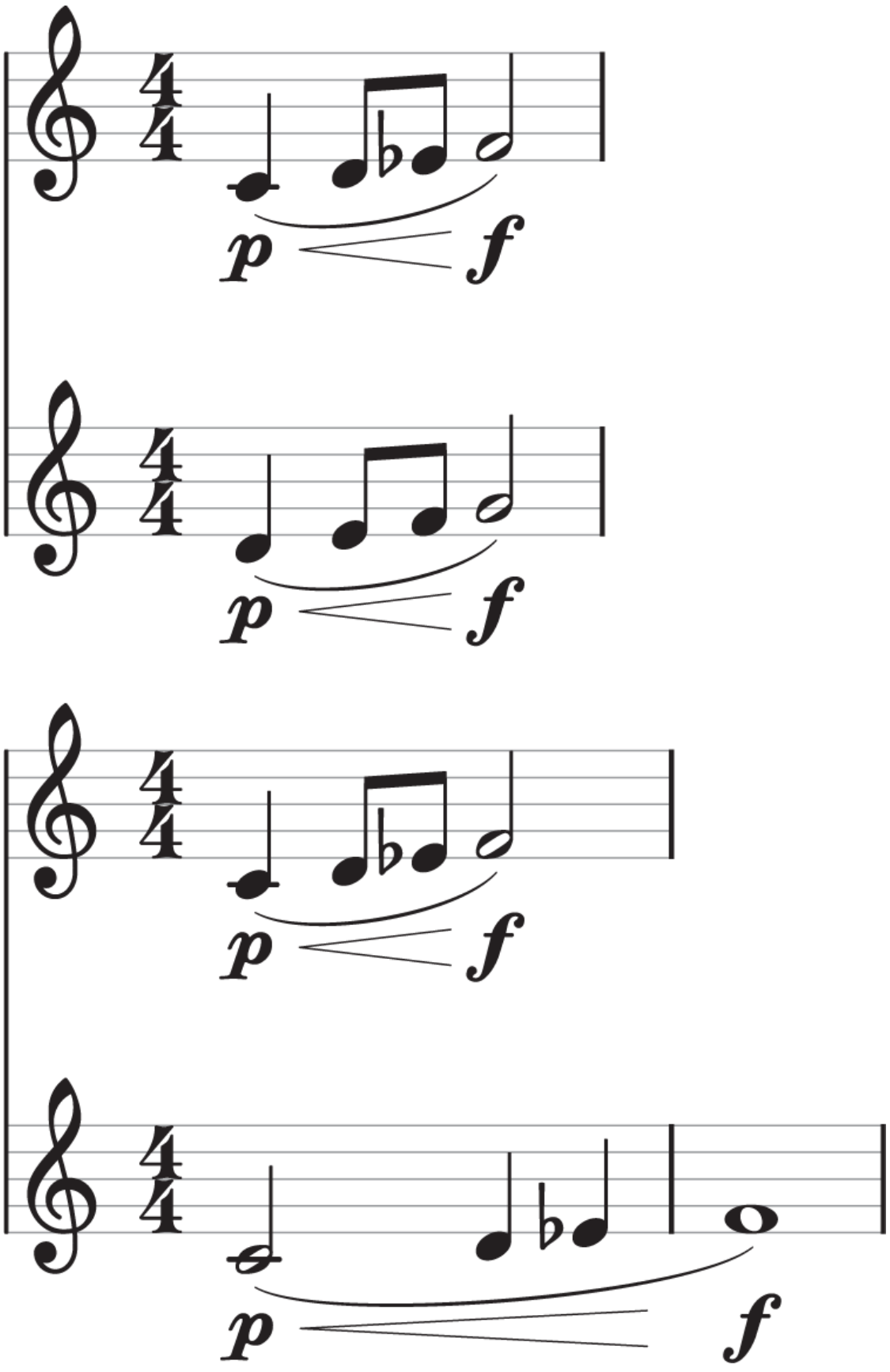



Figure 26.21
Results of code in figure 26.20.

26.4.3 Making sound

`Fosc` score objects maintain an internal representation of note, duration, and amplitude parameters as a list of Events, accessible through an *eventList* property. Score objects can be played directly in scsynth, as shown below using the default settings:

```
s.waitForBoot {FoscChord(#[60,64,67], 1/2).play};
```

The following shows how a sequence of notes may be played back using the default settings. Note how the musical dynamics are reflected in playback:

```
(
a = FoscStaff((60..72).collect {|pitch| FoscNote(pitch, 1/16)});
a.selectLeaves.hairpin('ppp < fff');
a.play;
)
```

The *eventList* property allows for the default settings to be transformed and/or overridden.

```
(
SynthDef('ping', {|midinote=60, amp=1|
   var envgen, src;
   envgen = EnvGen.kr(Env.perc(0.01, 0.5), doneAction: 2);
   src = Pan2.ar(SinOsc.ar(midinote.midicps, 0, amp), 0);
   OffsetOut.ar(0, src * envgen);
}).add;
)
(
a = FoscStaff((60..72).collect {|pitch| FoscNote(pitch, 1/16)});
a.selectLeaves.hairpin('ppp < fff');
Pbindf(Pseq(a.eventList), \instrument, 'ping').play;
)
```

Figure 26.22
Provide a custom SynthDef for playing a FoscStaff.

26.4.4 High-level interface: FoscMusicMaker

`FoscMusicMaker` is a factory class for the high-level algorithmic specification of musical materials. It encapsulates the more granular detail of `Fosc`'s low-level classes, allowing composers to focus on the musical parameters that are generally of the most

immediate interest when composing with standard musical notation: durations, rhythms, and pitches.

`FoscMusicMaker` follows a two-step configure-once / call-repeatedly pattern. At the configuration stage, preferences can be specified for beaming and the spelling of durations and rhythms. A `FoscMusicMaker` instance is called using the `value` method (the following examples use the method shortcut in the form `a.(args)`). The same instance can be called repeatedly to produce multiple musical variants from distinct sets of parameters. In figure 26.23, an instance of FoscMusicMaker is called with *durations* and *pitches*. The sequence of *pitches* repeats cyclically to match the length of the sequence of *durations*.

```
(
a = FoscMusicMaker();
b = a.(durations: [1/4, 1/8, 1/8, 1/2], pitches: #[60,62]);
b.show;
)
```

Figure 26.23
Simple usage of `FoscMusicMaker`.

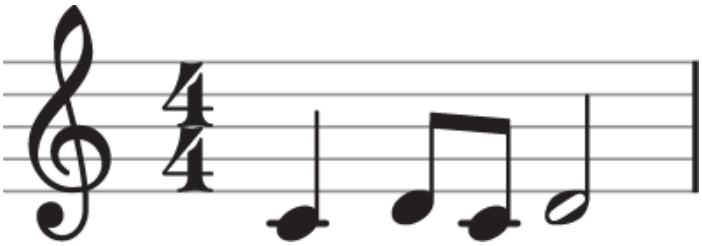



Figure 26.24
Result of code in figure 26.23.

The basic *durations* and *pitches* usage of `FoscMusicMaker` is suitable to musical situations based on simple *additive* rhythmic structures. More complex *divisive* rhythmic structures can be realized using the *divisions* argument in calls to a `FoscMusicMaker` instance, as shown in figure 26.25. *Divisions* are rhythmic ratios that embed into *durations*.

```
// Embed divisions into durations. Negative divisions are interpret
ed as rests
(
a = FoscMusicMaker();
b = a.(durations: [1/2, 1/8, 3/8], divisions:   #[[-1,3],[3,2],[2,
2,-3]], pitches: #[60,62]);
```

```
b.show;
)
// Divisions can be nested using a variant of rhythm-tree syntax
(
a = FoscMusicMaker();
b = a.(durations: [1/2, 1/8, 3/8], divisions:   #[[-1,3],[3,2],[2,
[2,[3,2]],-3]], pitches: #[60,62]);
b.show;
)
```

Figure 26.25
`FoscMusicMaker` used to create *divisive* rhythmic structures.

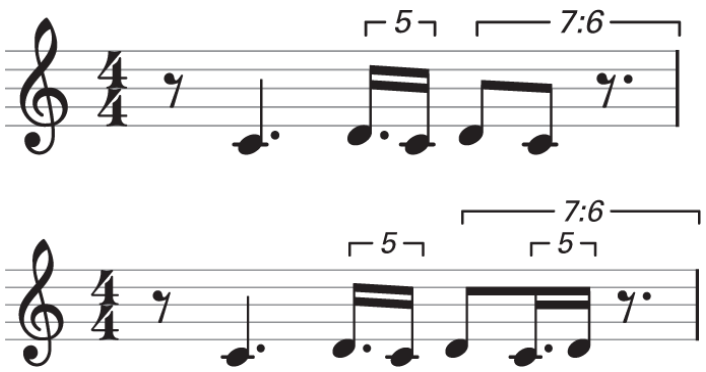



Figure 26.26
Results of code in figure 26.25.

Patterns can be "etched" into musical sequences using the *mask* argument in calls to a `FoscMusicMaker` instance. As shown in figure 26.27, when a *mask* is applied, contiguous musical events are fused in groupings that correspond to the group sizes in the *mask* array.

```
// An unmasked periodic sequence. The sequence of divisions repeats
cyclically to match the length of the sequence of durations
(
a = FoscMusicMaker();
b = a.(durations: 1/4! 4, divisions: #[[1,1,1,1]], pitches:   #[60,
62]);
b.show;
)
// The same sequence with a [2, 1] mask applied. The mask pattern r
epeats cyclically. Pitches are added after the mask is applied.
(
a = FoscMusicMaker();
b = a.(durations: 1/4! 4, divisions: #[[1,1,1,1]], mask:   #[2,1],
```

```
 pitches: #[60,62]);
b.show;
)
// The same sequence with a [2, -1] mask applied. Negative mask val
ues are interpreted as rests.
(
a = FoscMusicMaker();
b = a.(durations: 1/4! 4, divisions: #[[1,1,1,1]], mask:   #[2,-1],
pitches: #[60,62]);
b.show;
)
```

Figure 26.27
Melodic sequences demonstrating the use of a *mask*

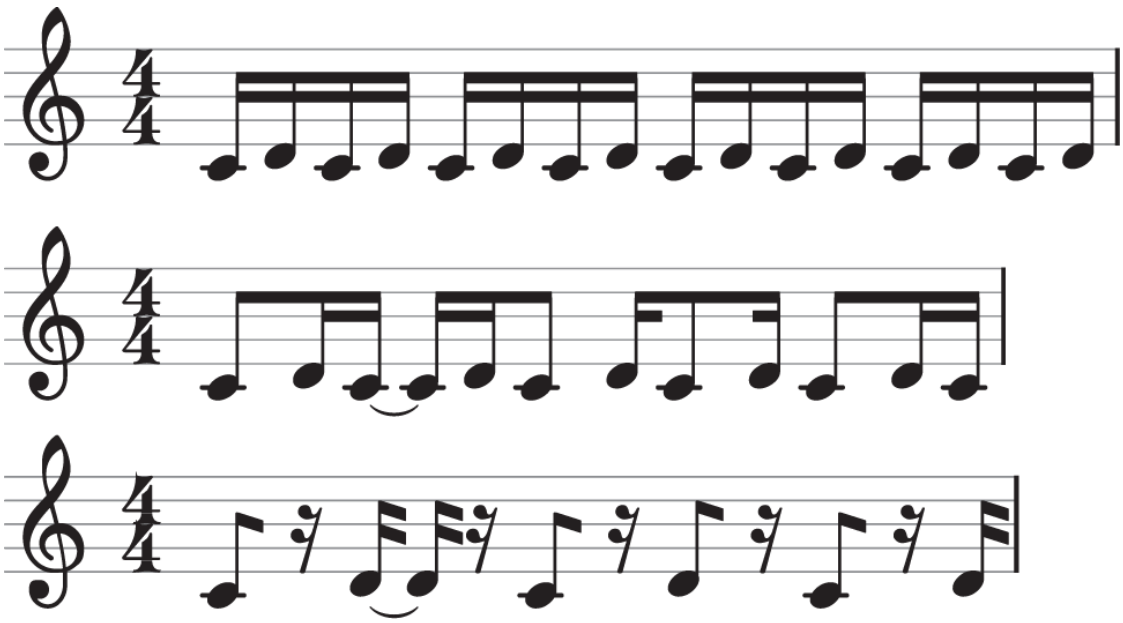



Figure 26.28
Results of code in figure 26.27.

Despite its simple interface, `FoscMusicMaker` provides a framework for complex and expressive specification of musical materials. Multiple variations on an algorithmic idea can be easily generated as standalone segments of music. The musical segments can be further decorated with indicators and spanners, allowing for a high level of specification and customization of notational detail.

26.4.5 FoscMusicMaker: An extended example

The interface provided by `FoscMusicMaker` is generic by design. It affords customized extensions—whether through subclassing or encapsulation—that are oriented towards

well-defined *types* of musical material. A compositional project may comprise many such types of material, and many variants of each type. Such an approach allows for compositional projects to evolve incrementally, defining new types of material along the way, while at the same time continuously refining the specification details of extant types. As new musical segments are generated through calls to individually customized `FoscMusicMakers`, they can be inserted into an expanding musical score.

A simple class, `InconjunctionsMaker`, is included as an example of how `FoscMusicMaker` can serve as a basis for customized extensions. The example in figure 26.29 calls `InconjunctionsMaker`'s `value` method to recreate a texture for a quartet of violins that appears at measure 103 of Brian Ferneyhough's *Inconjunctions* (2014).

```
(
~maker = InconjunctionsMaker();

~selections = ~maker.(

   durations: #[[4,8]],
   // divisions: the rhythmic ratio/s that embed/s into durations
   divisions: #[3,2,1,2].wrapExtend(24),
   // groupSizes: per voice segmentations
   groupSizes: #[
        [5,5,5,5,4],
        [6,6,6,6],
        [5,6,7,6],
        [3,4,5,6,6]
   ],
   // pitches: per voice ordered pitch sequences that repeat cyclic
ally at each segment
   pitches: #[
        "gqf'' bf'' cqf''' d''' ef''''",
        "d' g' bqs' aqf'' bf'' bqf''",
        "aqf d' af' bqs' gqf'' af'' bqf''",
        "gqs aqf d' g' af' bqs'"
   ],
   // hairpins: per voice, repeating cyclically at each segment
   hairpins: #['fff > f'],
   // articulations: per voice, applied cyclically to the first eve
nt in each segment
   articulations: #['>'],
   // finalize: ad hoc per voice modifications
   finalize: {|sel, i|
        // override default spelling of tuplet ratio
        sel.selectComponents(FoscTuplet).do {|tuplet|
```

```
                tuplet.denominator = 4;
                tuplet.forceFraction = true;
            };
        };
    );


    // collect selections of music into a score and add final details
    ~score = FoscScore(~selections.collect {|sel|   FoscStaff([FoscVoic
    e(sel)])});
    ~score.selectComponents(FoscStaff).do {|staff, i| set(staff)   .ins
    trumentName = "Vln. %".format(i + 1)};
    ~score.leafAt(0).attach(FoscTimeSignature(#[4,8]));
    ~score.show(staffSize: 14);
    )
```

Figure 26.29
A customized extension of `FoscMusicMaker`: `InconjunctionsMaker`.

There would be little to be gained from creating a custom class if one were interested only in producing the single measure of music in figure 26.30. The power of customized factory classes that extend `FoscMusicMaker` lies in their capacity to produce varied populations of musical material. For instance, a wide range of musical outcomes can be generated by calling `InconjunctionsMaker` with algorithmic variations on any or all of the *durations*, *divisions*, *groupSizes*, *pitches*, *hairpin*, and *articulations* parameters. One could even traverse the multi-dimensional parametric space combinatorially, filtering the output populations by user-specified preference rules.

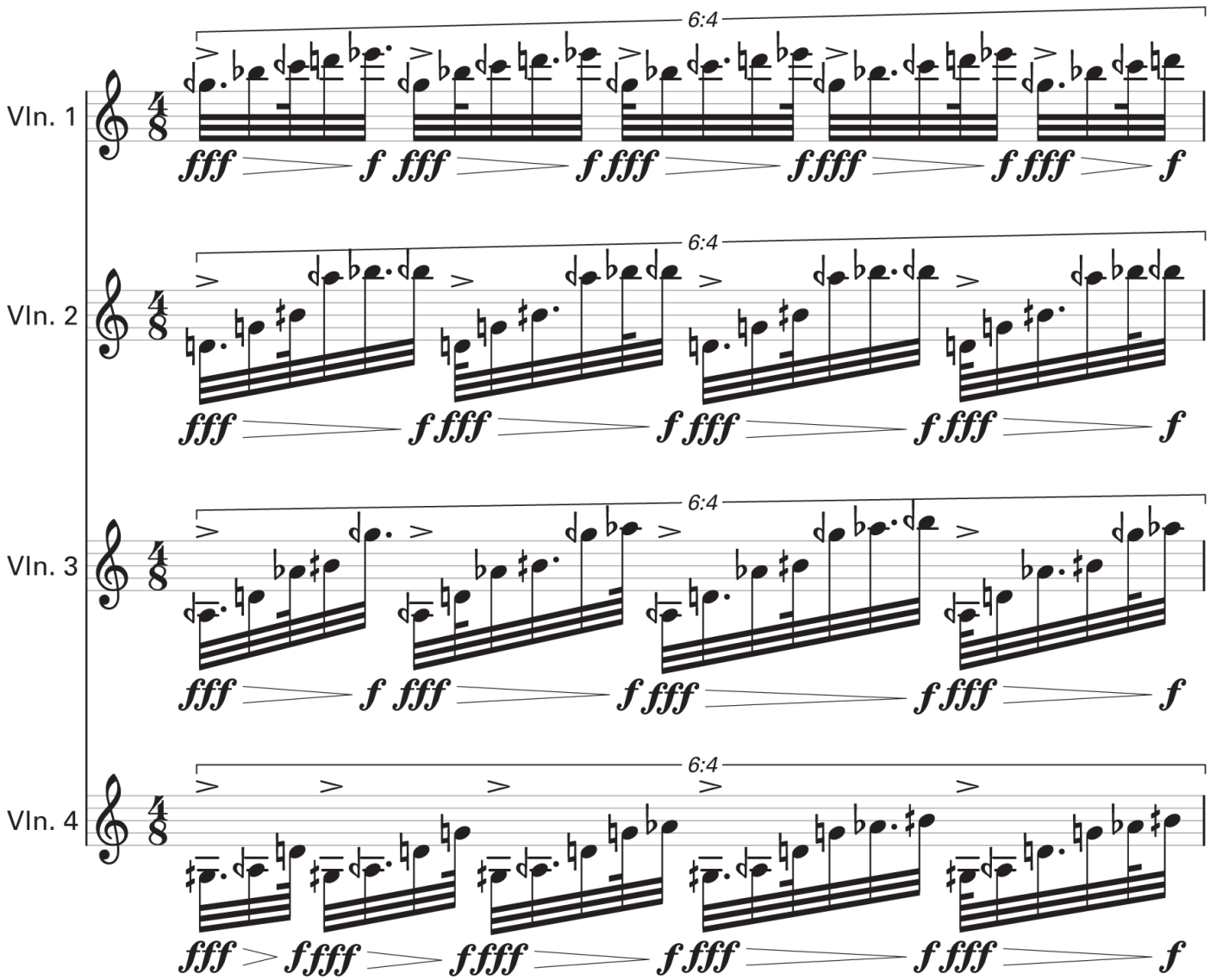



Figure 26.30
Results of code in figure 26.29.

26.5 Concluding Thoughts

As practitioners involved with different types of music notation, we believe different forms of musical notation broaden the possibilities of sonic thought and its expression. In support of this idea, we have created different interactive computer-assisted composition and notational environments using a wide variety of notations, fixed, algorithmic, "graphic" and traditional and extended CPWN.

For sound practitioners working with SuperCollider, the integration of notational tools into ways of working can leverage fluency and expressivity gained through programming experience into their sonic work. Whether using SuperCollider on its own or in conjunction with other specialist notational software, we have suggested that tight integration between approaches to the sonic and the visual creates benefits for both live

and fixed work involving notation. We are aware that each of the projects in this chapter presents certain musical and notational affordances as well as constraints. Each brings styles of interaction to the fore that we hope may suggest new and effective approaches to compositional thinking with sound and its visual prescriptions and representations.

Notes

1. In covering this notational territory we acknowledge the wide pedigree of the work of others in using SuperCollider in the context of music notation as here described, including that of Nick Collins and Fredrik Olofsson (see Sauer, 2009, p.56), Rohan Drape, John Eacott, Thor Magnusson, Josh Parmenter, and Andrea Valle, among others. See also responses at https://scsynth.org/t/do-you-make-visual-scores-music-notation-using-sc/5219
2. For an example of another type of SuperCollider live graphic notation by one of the authors, see Hall & Blackwell, 2014.
3. The relevant SuperCollider classes for this section can be downloaded from https://github.com/ludions/scbook-2nd-edn
4. UTF-8 music font characters can be freely cut and pasted, e.g. from the Internet, however won't display in SuperCollider's IDE unless the font is installed and the editor set to that font; see the `String` help file.
5. See http://www.unicode.org/versions/latest
6. https://w3c.github.io/smufl
7. https://www.smufl.org/fonts
8. https://w3c.github.io/smufl/latest/tables/clefs.html
9. https://github.com/w3c/smufl
10. https://docs.cycling74.com/max8/refpages/nslider
11. See the neoscore project for a related approach in the recreation of the score of Feldman's *Projection #2*. Available at https://github.com/DigiScore/neoscore
12. A classic early text discussing this general approach, as well as an example of Feldman's graph scores is Behrman, 1965.
13. https://inscore.grame.fr
14. See https://inscoredoc.grame.fr
15. You can experiment with this using the `INScoreListenerMIT` class.
16. The relevant SuperCollider classes for this section can be downloaded from https://github.com/richardhoadley/inscoremit
17. https://guidodoc.grame.fr. Discussion of INScore's MusicXML engine is beyond the scope of this chapter.
18. https://github.com/grame-cncm/inscore/releases
19. On macOS, applications that do not have focus can be slow to reflect changes. The osaActivate method shown in the current example will bring INScore to the foreground on macOS, which should solve any delays in updating changes via communication with SuperCollider. Note that for communication with secondary computers, INScore must first be opened for the method to work. The osaActivate method is ignored by other operating systems.
20. See for example https://inscoredoc.grame.fr/refs/6-setsect/#vectorial-graphics
21. In INScore the native code for this is `ITL/scene/polygon0 set poly [0.7, 0.05, 0.1, 0.1, 0.1, 0.0]];`
22. Whether audience members are able to read CPWN or not, audience reaction to the public screen display has been positive; feedback has generally indicated an interest in the appearance and behavior of the varieties of notation displayed.
23. The composition's title and much of the text used is drawn from George Berkeley's *A Treatise Concerning the Principles of Human Knowledge* (Berkeley: 1710), in which the author explores the distinction between thinking and "unthinking" things—inanimate objects.
24. Texts are obtained algorithmically from Berkeley's *Treatise* and from algorithmically generated haiku, the available vocabulary focusing on elements of Berkeley's philosophical interests.
25. A rendition of this can be seen at: https://youtu.be/64HImuOHtpc?t=734, which links to a recording of a performance from 17th April 2021. This performance was held online due to the prevailing international health

conditions. All rehearsals and performances took place from the singers' private residences with networked video, audio and data.

26. A rendition of this section can be seen and heard here: https://youtu.be/64HImuOHtpc?t=319. In a 1986 documentary about Cornelius Cardew, produced by the Arts Council of Great Britain (Regniez 1986), Eddie Prévost provides an entertaining account of a time when members of the improvisation group AMM were encouraged by Cardew to search a particular area within the UK's Peak District for examples of "musical stones" for use in performance. Performers might be encouraged to use similar motivations when locating the materials used for performance here.

27. https://youtu.be/64HImuOHtpc

28. https://youtu.be/64HImuOHtpc?t=374

29. https://lilypond.org

30. https://abjad.github.io

31. https://github.com/n-armstrong/fosc

References

Armstrong, N. 2018. "Thread—surface." in Armstrong, N., 2019. *The way to go out* [CD]. Another Timbre, at167.

Armstrong, N. 2019. "A line alongside itself." in Armstrong, N. 2019. *The way to go out* [CD]. Another Timbre, at167.

Behrman, D. 1965. "What Indeterminate Notation Determines." *Perspectives of New Music*, vol 3, no. 2: 58–73.

Berkeley, G. 1710. *A Treatise Concerning the Principles of Human Knowledge.* Dublin: Jeremy Pepyat. Available online: https://www.gutenberg.org/ebooks/4723

Bringhurst, R. 1999. *The Elements of Typographic Style.* 2nd Edn. Point Roberts, WA: Hartley & Marks.

Brown, E. 2008. "On December 1952." *American Music*, vol 26, no. 1: 1–12.

Cardew, C. 1971. *The Great Learning*, London: Experimental Music Catalogue.

Cline, D. 2017. *The Graph Music of Morton Feldman*. Music since 1900. Cambridge: Cambridge University Press.

Ferneyhough, B. 2014. *Inconjunctions*, for chamber orchestra. Leipzig: Edition Peters.

Gamma, R. et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

Hall, T. and A. F. Blackwell. 2014. "Sharing digital performance notation with the audience." In *Proceedings of the 9th Conference on Interdisciplinary Musicology—CIM14*, Staatliches Institut für Musikforschung, Berlin. Available online: http://www.ludions.com/texts/

Higgins, H. 2009. *The Grid Book*. Cambridge MA: MIT Press.

Hiller, L. A., and L.M. Isaacson. 1959. "Experimental Music / Composition with an Electronic Computer." New York: McGraw Hill.

Hoadley, R. 2017. "Homenaje a Cervantes: For Violin, Computer And Projections." In *Proceedings of the International Conference on Technologies for Music Notation and Representation* (TENOR). A Coruña, Spain. Available online: https://hal.archives-ouvertes.fr/hal-03165874/file/27_Hoadley_tenor2017.pdf

Iverson, K. E. 1980. "Notation as a tool of thought." *Communications of the ACM* 23, no. 8 (Aug): 444–465.

Mathews, M. V., and L. Rosler. 1968. "Graphical Language for the Scores of Computer-Generated Sounds." *Perspectives of New Music* 6, no. 2: 92–118.

Regniez, P. 1986. *Cornelius Cardew 1936–1981* [Film]. Arts Council of Great Britain. Available online: https://player.bfi.org.uk/free/film/watch-cornelius-cardew-1936-1981-1986-online

Sauer, T. 2009. *Notations 21*. New York: Mark Batty.

Schnebel, D. 1969. "Visible Music." In *Classic Essays on Twentieth-Century Music / A Continuing Symposium*, ed Kostelanetz, R., Darby, J. Santa, M., 1996. New York: Schirmer Books.

Veer, C. 2019. *The Digital Score / Musicianship, Creativity and Innovation*. New York: Routledge.

Weibel, P. 2020. “The Road To The Upic. From Graphic Notation To Graphic User Interface.” In Weibel, Brümmer, Kanach, eds., 2020. *From Xenakis's UPIC to Graphic Notation Today*. Berlin: Hatje Cantz. Available online: http://www.interfacesnetwork.eu/post.php?pid=47-upic-book

Wright, M., and A. Freed. 1997. “Open SoundControl: A New Protocol for Communicating with Sound Synthesizers.” In *Proceedings of the International Computer Music Conference* (ICMC): 101–104.

27 SCTweets: Character Matters

Fellipe M. Martins

In March 2009, Dan Stowell posted a short example of fully runnable SuperCollider (SC) code on Twitter,[1] then a fairly recently created—but rapidly growing—microblogging website on which users' interaction took place within a 140-character limit (Stowell, 2009). Shortly after, a discussion of this arose on the SuperCollider mailing list (SaRiGaMa's Oil Vending Orchestra, 2009), the central communication mean adopted by the SuperCollider community at that time—and several users joined in the game.[2] A couple of months later, this interaction resulted in a collective album of 140-character "SCTweets" recordings (Stowell et al. 2009). The SuperCollider tweeting practice was thus born and established. Within it, you can find drones, full compositions, minimalist pieces, dance music, an enormous range of noise flavors and random processes, soundscapes, metapoetic code, generative art, and ASCII animations on the post window.

Throughout this chapter, we will focus on a selection of SCTweets that explore unexpected procedures, usages, and results of 140-character SC programs. We aim to demonstrate how the imposition of a hard constraint like this character limit has inspired users to adopt creative and unexpected synthesis strategies that yield complex and diverse sonic output.

27.1 Code Optimized for Length—How Expressive Can Laconic Code Be?

The SCTweet practice is closely connected to so-called code golfing, a game/competition where participants seek to solve computer problems by using the shortest possible bits of code.[3] Despite its different goals, SCTweeting can also be associated with code *minification* (reducing size and removing unnecessary code characters without changing functionality) and code *obfuscation* (modifying source code to decrease human readability and understanding). Both serve as industry techniques for improving information security.

The practice of optimizing code for length often leads to the poetic and humorous concept of "write-only code," where programming expressions become so dense and

complex that they are almost unreadable. Moreover, SCTweets generally incorporate some kind of metapoetical reference or programming humor as part of an effort to write condensed code, such as SCTweets that iterate through the whole UGen class library, playing each with default arguments; rearranging the playback of the default audio sample to pronounce the word "SuperCollider"; the sonification of textual data to produce auditory outputs related to the text meaning; and others.

As a high-level language for musical and sonic applications, SC offers a considerable number of approaches for performing complex tasks using short expressions. Moreover, it usually allows several syntax possibilities for implementing the same task.[4] Figure 27.1 shows a few language shorteners that can be used for adapting SC code to a short character limit.[5]

```
// Usage of Syntax Shortcuts
Routine{"x".postln;1.wait;"y".postln}.play;
r{"x".postln;1.wait;"y".postln}.play;

// Array Creation Shortcuts
{SinOsc.ar([1,3,5,7,9,11,13]*100).sum/8}.play;
{SinOsc.ar((1,3..13)*100).sum/8}.play;

// Partial Application
play{Mix(SinOsc.ar((0..100)))};
play{Mix(SinOsc.ar(_)!100)};

// Receiver vs Functional vs Binary Operator notations

rrand(2,9) // Functional Notation
2.rrand(9) // Receiver Notation
2 min: 9 // Binary Operator Notation

{SinOsc.ar(440)}.play;
play{SinOsc.ar(440)};

play{ar(SinOsc,440)};
play{SinOsc.ar(440)};
play{SinOsc ar:440};

// Sucessive Chained Methods
{x=SinOsc.ar(440);x=x.softclip;x=x.cubed;x=x.rand}.play;
play{SinOsc.ar(440).softclip.cubed.rand};

// Assign UGens to Variables
{SinOsc.ar(440)/2*SinOsc.kr(1)}.play;
```

```
x=SinOsc;{x.ar(440)/2*x.kr(1)}.play;

// Assign Variables Inside Statements
play{x=PinkNoise.ar();BLowPass4.ar(x*2)*x};
play{BLowPass4.ar(2*x=PinkNoise.ar)*x};

// Left to Right Order of Precedence
1 + 2*3
3*(1 + 2)

play{x=SinOsc;x.ar(1e3*(x.ar(5)+1))};
play{x=SinOsc;x.ar(1+x.ar(5)*1e3)};
```

Figure 27.1
Syntax techniques used in SCTweets.

While a high intake of syntactic "sugar" is crucial for sweetening the code pill, heterodox synthesis and DSP techniques also play a special role for achieving complexity and nonstandard results. Some recurrent examples are the use of phase modulation feedback to obtain sinusoids, triangle waves, and white noise with a single oscillator (e.g., `SinOscFB`); the practice of adding square waves closely spaced in frequency to obtain a stepper function characterized by a high degree of variability; and the use of binary numbers and operators as rhythmic generators/transformations. It is common to see Bytebeat (i.e., bitwise operations on audio signals producing rhythms and timbres all at once), intentional aliasing to create formants or filtering, applications of general mathematical and computing procedures to audio and musical data (hash function, ASCII conversion, statistical measures, and functions such as gamma and Bessel).

In addition, unsafe audio procedures are often used to reduce the code size, including clipping due to excessive amplitude levels, the introduction of DC to modify low-frequency oscillations, intentional induction of server crashes to produce audio glitches, and aliasing to produce a creative effect. Despite the risk involved in executing these procedures, they generally tend to produce idiosyncratic sonic results. When playing SCTweets, be careful with volume, use limiters (e.g., StageLimiter Quark) and be prepared for loud, high, or unpredictable results.

27.2 Self-Regulating Scottish Raga

Producing interesting results within the 140-character limit is a difficult task. Adding to this a degree of agency[6] and interaction[7] is also challenging because it often demands some type of analysis which can be excessively verbose or introduce undesired noisy,

chaotic, or extremely complex behaviors. Figure 27.2 shows code created by Nathaniel Virgo, named "Scottish Raga,"[8] which implements a self-regulating system that plays melismatic notes over a drone, with a timbre that resembles a Scottish bagpipe (Virgo 2012a, 2012b).

```
play{ar(r=RLPF,Saw.ar([200,302]).mean,5**(n={LFNoise1.kr(1/8)})    *
1e3,0.6)+r.ar(Saw ar:Amplitude.kr(3**n*3e3*InFeedback.    ar(0)+1,4,
4),1e3)/5!2}

// 1 - Regular Indentation
(
{
   ar(
        r=RLPF,
        Saw.ar([200,302]).mean,5**(n={LFNoise1.kr(1/8)})*1e3,
        0.6
   )
   +r.ar(
        Saw ar:Amplitude.kr(3**n*3e3*InFeedback.ar(0)+1,4,4),
        1e3
   )/5!2
}.play;
)

// 2-Separate Variable Definitions, Functional/Receiver Notation Ch
anges
(
{
   r=RLPF;
   r.ar(
        Saw.ar([200,302]).mean,
        5**(n={LFNoise1.kr(1/8)})*1e3,
        0.6
   )
   +r.ar(
        Saw.ar(
              Amplitude.kr(
                   3**n*3e3*InFeedback.ar(0)+1,
                   4,
                   4)
        ),
        1e3
```

```
    )/5!2
}.play;
)

// 3 – Explicit Argument
(
{
    RLPF.ar(
        in: Saw.ar(freq:[200,302]).mean,
        freq: 5**(LFNoise1.kr(1/8))*1e3,
        rq: 0.6
    )
    +RLPF.ar(
        in: Saw.ar(
        freq: Amplitude.kr(
            in: 3**LFNoise1.kr(1/8)*3e3*InFeedback.ar(0)+1,
            attackTime: 4,
            releaseTime: 4)
        ),
        freq: 1e3
    )/5!2
}.play;
)
```

Figure 27.2
"Scottish Raga" by Nathaniel Virgo.

Figure 27.2 also demonstrates an approach for decoding SCTweets involving the following steps: (1) reformat with traditional indentation; (2) explicitly define all the variables and swap receiver/functional notation to improve readability; and (3) convert to keyword argument style. After performing these steps, we can see that the code consists of two summed oscillators (`LFSaw.ar`), both filtered and frequency-modulated individually.

Additionally, the melodic oscillator's frequency is determined by the amplitude of the output in a feedback process. Low-frequency random modulations and beating effects introduce slow variations into the drone resonances and the melodic oscillator pitch curve. An intricate calculation process due to the analysis lag and control rate use is also responsible for producing the steady pitch steps that are heard (Virgo, 2012b).

27.3 Recursive Modulation of Frequency, Phase, and Amplitude

Fredrik Olofsson might be considered the most prolific SCTweeter, regularly posting SC code and reusing it for installations, live coding, generative pieces, and other

activities (Olofsson, 2020). He has been exploring a large variety of SCTweets, such as ASCII graphics and animations, generative plots, language/server communication, and restriction to a single numeric value or a single UGen and wordplays.

One of his one-liners deals with the use of recursion to produce a dense texture/drone using solely sinusoidal oscillators, as shown in figure 27.3. On the whole, function `f` defines a sinusoidal oscillator where the frequency, phase, and amplitude are modulated by another sinusoid also defined by `f`, one recursive step further. The recursion goes on until it reaches the terminating scenario, where function `f` has a numerical value defined by argument `o`. After reaching this base case, all the self-defined sinusoidal oscillators can be unfolded and finally be numerically determined. The client code expressions for producing these oscillators would be extremely long, as shown in figure 27.4. In this regard, recursion facilitates the implementation of complex manifold and fractal processes.

```
//—tweet0011
play{f={|o,i|if(i>0,{SinOsc.ar([i,i+1e—4]**2*f.(o,i—1),    f.(o,i—1)*
1e—4,f.(o,i—1))},o)};f.(60,6)/60}//#SuperCollider

// 1 — Regular Indentation And Explicit Arguments
play{
    f={|o,i|
        if(i>0,
            {SinOsc.ar(
                freq: [i,i+1e—4]**2*f.(o,i—1),
                phase: f.(o,i—1)*1e—4,
                mul: f.(o,i—1))
            },
            {o}
        )
    };
    f.(60,6)/60
}
```

Figure 27.3
Recursive modulation of frequency, phase, and amplitude, by Fredrik Olofsson.

Such a short and dense code produces an enormous synthesis graph which requires an impractical amount of space to be represented, as shown by Olofsson (2012). It reveals a peculiar potential of SuperCollider: users are not restricted to interaction with static objects. On the contrary, they can define objects that can define other objects and behaviors, a chain reaction of computer music processes. When dealing with visual

programming languages alternatively, the implementation of such a paradigm seems unattainable due to the risk of sacrificing visibility and readability.

```
f(o,i) = SinOsc.ar(freq: [i,i+1e−4]**2*f.(o,i−1), phase:   f.(o,i−
1)*1e−4, mul: f.(o,i−1))
i = 60, f(6,60):
SinOsc.ar(freq: [60,60 + 1e−4]**2*f.(6,59), phase: f.(6,59)*1e−4,
mul: f.(6,59))

i = 59, f(6,59):
SinOsc.ar(freq: [59,59 + 1e−4]**2*f.(6,58), phase: f.(6,58)*1e−4,
mul: f.(6,58))
. . .
. . .
. . .
i = 3, f(6,3):
SinOsc.ar(freq: [3,3 + 1e−4]**2*f.(6,2), phase: f.(6,2)*1e−4,   mu
l: f.(6,2))

i = 2, f(6,2):
SinOsc.ar(freq: [2,2 + 1e−4]**2*f.(6,1), phase: f.(6,1)*1e−4,   mu
l: f.(6,1))

i = 1, f(6,1):
SinOsc.ar(freq: [1,1 + 1e−4]**2*f.(6,0), phase: f.(6,0)*1e−4,   mu
l: f.(6,0))

i = 0 −> i>0 == false
f(6,0) == o == 6.
terminating scenario.

then we proceed to solving each high order recursive step:

i = 1, f(6,1):
SinOsc.ar(freq: [1,1 + 1e−4]**2*6, phase: 6*1e−4, mul: 6)

i = 2, f(6,2):
SinOsc.ar(freq: [2,2 + 1e−4]**2*SinOsc.ar(freq: [1,1 + 1e−4]**2*6,
phase: 6*1e−4, mul: 6), phase: SinOsc.ar(freq: [1,1 + 1e−4]**2*6,
phase: 6*1e−4, mul: 6)*1e−4, mul: f.(6,1))
```

```
i = 3, f(6,3):
SinOsc.ar(freq: [3,3 + 1e−4]**2*SinOsc.ar(freq: [2,2 + 1e−4]   **2*
SinOsc.ar(freq: [1,1 + 1e−4]**2*6, phase: 6*1e−4, mul: 6),   phase:
SinOsc.ar(freq: [1,1 + 1e−4]**2*6, phase: 6*1e−4,   mul: 6)*1e−4, m
ul: f.(6,1)), phase: f.(6,2)*1e−4, mul:   SinOsc.ar(freq: [2,2 + 1e
−4]**2*SinOsc.ar(freq: [1,1 + 1e−4]**2*6,   phase: 6*1e−4, mul: 6),
phase: SinOsc.ar(freq: [1,1 + 1e−4]**2*6,   phase: 6*1e−4, mul: 6)*
1e−4, mul: f.(6,1)))
. . .
. . .
. . .
```

Figure 27.4
Analytical expressions of Olofsson's recursive SCTweet.

27.4 Quine: Quine

So far, we can notice that one of the key strategies for writing a complex SCTweet is reusability. This can be taken to its limits when using *quines*, computer programs that take no input and produce a copy of their own source code as their only output (Wikipedia 2021). Although writing a noncheating quine under 140 characters is either extremely difficult or sonically too simple, SCTweets examples that use the array's `.interpret` method can be considered quinelike programs.[9]

On the tweet shown in figure 27.5, by Nathan Ho, the major part of the code is defined as a string which is used to execute both the client language command and a converter to an array of ASCII values which will fill a buffer (Ho, 2014). This buffer is going to be read through interpolating and chorusing wavetable oscillators (`COsc` and `Osc`), performing a sonification of the own source code.

```
interpret(x="/******/play{RLPF.ar(COsc.ar(b=Buffer.loadCollection
(s,x.ascii.normalize(−1,1)),Osc.kr(b,1,0,65,35)),Osc.kr(b,0.1,   0,
300,1e3))/5}")

// 1 − Regular Indentation, Functional/Receiver Notation Changes
(
x="/******/play{RLPF.ar(COsc.ar(b=Buffer.loadCollection(s,x.   asci
i.normalize(−1,1)),Osc.kr(b,1,0,65,35)),Osc.kr(b,0.1,0,300,   1e
3))/5}";
{
   RLPF.ar(
        COsc.ar(
```

```
                b=Buffer.loadCollection(s,x.ascii.normalize(-1,1)),
                Osc.kr(b,1,0,65,35)
            ),
            Osc.kr(b,0.1,0,300,1e3)
        )/5
}.play;
)

// 3 - Explicit Arguments
(
x="/******/play{RLPF.ar(COsc.ar(b=Buffer.loadCollection   (s,x.asci
i.normalize(-1,1)),Osc.kr(b,1,0,65,35)),Osc.   kr(b,0.1,0,300,1e
3))/5}";
{
    RLPF.ar(
        in: COsc.ar(
            bufnum: Buffer.loadCollection(s,x.ascii   .normalize(
-1,1)),
            freq: Osc.kr(
                bufnum: Buffer.loadCollection(s,x.ascii   .norma
lize(-1,1)),
                freq: 1, phase: 0, mul: 65, add: 35
            )
        ),
        freq: Osc.kr(
            bufnum: Buffer.loadCollection(s,x.ascii   .normalize(
-1,1)),
            freq: 0.1,phase: 0,mul: 300,add: 1e3
        )
    )/5
}.play;
)
```

Figure 27.5
Quine, by Nathan Ho.

27.5 Final Remarks

Overall, using more compact code leads to an increase in sound and computing complexity because it is often obtained by an increase of abstraction. Accordingly, the sound results of such small pieces of code tend to be quite difficult to predict, leading to exquisite, dangerous, or intriguing surprises. In this delicate balance, SCTweets bear a resemblance to aphorisms and haikus, compressing numerous sonic possibilities into a

dense text in which the association between key words and superstitious numbers promotes an aural meaning to be meditated by the listener.

Notes

1. Now called X. As this chapter looks at the historical practice, we will use the terms Twitter/Tweet.
2. On Twitter, you can search for the tags #SuperCollider, #SC, and expressions like "play{," and "SinOsc."
3. See https://codegolf.stackexchange.com/, in particular the "Showcase of Languages" section.
4. From the early versions of SuperCollider, James McCartney's examples have demonstrated considerable terseness and brevity, referred to as "haiku programming" by Alberto de Campo. See, for example, http://web.archive.org/web/20030506224304/http://www.audiosynth.com/schtmldocs/Tutorials/SC2_Tutorial_0.8.5/3ProgrammingTechniques/3.3PatchanalysisWhy.html.
5. Based and adapted from https://schemawound.com/2012/06/23/supercollider-tweets-background-tips/.
6. The system's ability to take action by itself (or to choose what action to take).
7. The possibility of the user/listener to change the program result while it is executed.
8. For an analysis by the author, see Virgo (2012b).
9. For some sonic quines examples, check the quines.scd in SC's examples folder.

References

Ho, Nathan (@snappizz). 2014. "interpret(x="/***/play{RLPF.ar(COsc.ar(b=Buffer.loadCollection(s,x.ascii.normalize(-1,1)),Osc.kr(b,1,0,65,35)),Osc.kr(b,0.1,0,300,1e3))/5}")" Sccode, March 10. https://web.archive.org/web/20170224050108/http:/sccode.org/1-4VV.

Olofsson, Fredrik. 2012. "One Reason Why I Love SC." *f0blog* (blog), May 8, https://fredrikolofsson.com/f0blog/one-reason-why-i-love-sc/.

Olofsson, Fredrik. 2020. "Cybernetic Music in Practice." *Sonic Ideas/Ideas Sónicas,* 12, no.23 (July–December 2020): 37–41. ISSN: 2317–9694.

SaRiGaMa's Oil Vending Orchestra. 2009. "sctwitt" sc-users (mailing list). *Chronicle of Higher Education*, March 28, http://www.listarc.cal.bham.ac.uk/lists/sc-users-2009/msg50708.html.

Stowell, Dan (@mclduk). 2009. "{x=Line.kr(0,[0.9,1],30);5.do{x=[Saw,SinOsc, Pulse].choose.ar(x.range(10, 1000))};x*0.1}.play; // #supercollider." Twitter, March 27. https://twitter.com/mclduk/status/1401081491.

Stowell, Dan, Nathaniel Virgo, Till Bovermann, et al. 2009. "sc140." October 20, http://supercollider.sourceforge.net/sc140 … /

Virgo, Nathaniel (@headcube). 2012. "play{ar(r=RLPF,Saw.ar([200,302]).mean,5**(n={LFNoise1.kr(1/8)})*1e3,0.6)+r.ar(Saw ar:Amplitude.kr(3**n*3e3*InFeedback.ar(0)+1,4,4),1e3)/5!2}." Twitter, November 28. https://twitter.com/headcube/status/273708223440244736.

Virgo, Nathaniel. 2012. "SC tweet from 28.11.2012." *SoundCloud*, November 28. https://soundcloud.com/nathaniel-virgo/sc-tweet-from-28-11-2012.

Wikipedia. "Quine." Last modified December 20, 2021. https://en.wikipedia.org/wiki/Quine_(computing).

VI DEVELOPER TOPICS

28 The SuperCollider Language Implementation

Stefan Kersten

28.1 Introduction

In this chapter, we will have a closer look at the SuperCollider (SC) compiler and interpreter, its internal data structures, coding conventions, and inner workings. The last part of the chapter is devoted to writing language primitives in C++, which is necessary for interfacing to external C libraries and operating system APIs.

The SuperCollider interpreter is a complete implementation of an object-oriented language in the Smalltalk tradition that was written from scratch by James McCartney for SuperCollider 2 (McCartney, 1996). While some aspects, such as byte-code interpretation, class hierarchy, and object layout, can be recognized in similar existing interpreters, other features, such as its orientation toward audio and music, real-time garbage collection, and constant-time method lookup, are still important in SuperCollider, and they were unique to the language when it was first released.

In general, there are four key aspects of the design and implementation of a virtual machine: the layout of objects in memory, the virtual machine instruction set, the compiler, and the interpreter or virtual machine itself. The following sections are intended to give you a broad overview of all these aspects and to enable you to dig deeper according to your interests and needs. To this end, it also would probably be beneficial to look at other virtual machine implementations and design documents (Goldberg and Robson, 1983; Budd, 1987; Ingalls et al., 1997).

28.2 Coding Conventions

Before delving into the implementation details, we will introduce some coding conventions used throughout the source code which will (hopefully) help you to better understand the implementation.

SuperCollider is based on C as the main implementation language, using object-oriented features from C++ where appropriate (e.g., when sharing data in related hierarchies of implementation structures). Due to the inherently low-level nature of interpreters, high-level features of the host language usually don't map well to features in the target language, so it comes as no surprise that SuperCollider's virtual machine

implementation was originally mostly restricted to pure C. Compilers, however, do benefit from high-level abstractions: that is why SuperCollider's byte-code compiler uses object-oriented features, such as inheritance and polymorphism, more intensively. As the language has grown, more C++ features have been adopted and some code converted to C++ style.

In the source code, there are a few naming conventions that can be identified: class, structure, and union type names are often written in camel case (http://en.wikipedia.org/wiki/Camel_case) and prefixed with a capital Pyr for historical reasons—the very first incarnation of SuperCollider was a *Max/MSP* object called *Pyrite*. Function and method names usually use camel case as well, but they start with a lowercase character (with some notable exceptions in low-level support code, such as the storage allocator and the garbage collector). Functions implementing primitives are prefixed with "pr"; global variables visible across modules often start with a lowercase "g"; and certain constants are prefixed with "k." Symbols (i.e., pointers to `PyrSymbol`) are usually prefixed with "s" or "s_."

Indentation is by tabs of width 4 throughout the source code; this convention should be adhered to when adding code in order to maintain a coherent code base.

28.3 Object Layout

The internal memory layout of objects is an important design consideration in any language implementation, as it constitutes the interface between compiler, interpreter, and garbage collector (figure 28.1). This section gives an overview of how objects (and pointers to them) are defined in SuperCollider and discusses some important object types in more detail.

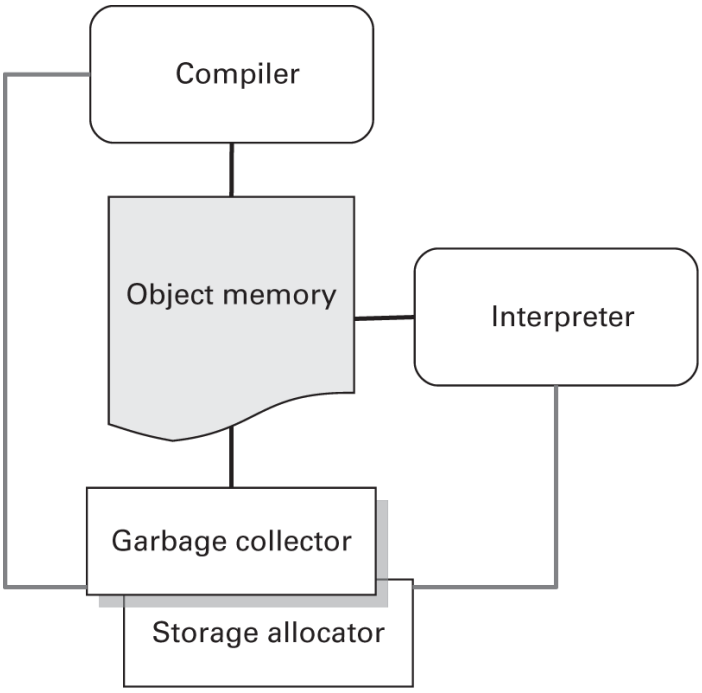


Figure 28.1
Overview of the SuperCollider language subsystems.

In dynamically typed, garbage-collected languages that don't permit accessing and storing references to objects in raw memory, objects need to be enriched with additional runtime information, such as their type, memory layout, and garbage collection state. Furthermore, it is often desirable to represent certain target language object types that see extensive usage—such as numeric values—not via a pointer to a heap-allocated object but directly encoded in a machine address. In SuperCollider, such tagged references reside in a special data structure called a slot (`PyrSlot`). On the older 32-bit machines on which SuperCollider was developed. this is bigger than a machine address, allowing additional information to be stored along with a machine pointer. On 32-bit machines, a `PyrSlot` is a `union` holding either a 64-bit IEEE double value or a 32-bit integer tag, followed by another union of 32-bit data (figure 28.2).

```
// 32 bit implementation; N.B. on little endian machines the   tag
 comes at the end of the union
Typedef union pyrslot
{
   double f; // double
   struct {
        int tag;
        union {
              int                  c;        // character
              int                  i;        // integer
              float                f;        // float
```

```
                void                    *ptr;      // raw pointer
                struct PyrObject        *o;        // object pointer
                PyrSymbol               *s;        // symbol pointer
                . . .                              // other object pointe
rs (see below)
        } u;
  } s;
} PyrSlot;

// 64 bit implementation
typedef struct pyrslot {
 long tag;

   union {
   int64 c; /* char */
   int64 i;
   double f;
   void* ptr;
   struct PyrObject* o;
   PyrSymbol* s;
   struct PyrMethod* om;
   struct PyrBlock* oblk;
   struct PyrClass* oc;
   struct PyrFrame* of;
   struct PyrList* ol;
   struct PyrString* os;
   struct PyrInt8Array* ob;
   struct PyrDoubleArray* od;
   struct PyrSymbolArray* osym;
   struct PyrProcess* op;
   struct PyrThread* ot;
   struct PyrInterpreter* oi;
} u;
} PyrSlot;
```

Figure 28.2
`PyrSlot` data structure.

This allows the slot to be used either as a double value, with no storage or dereferencing overhead, or as a tagged value with data contained in the lower 32 bits of the 64-bit double word (figure 28.3). The tag is stored in the unused high-order bits of an IEEE not a number double (IEEE Standards Board, 1985), so as not to limit the range of useful floating-point values. All the tags are defined and their uses listed in table 28.1. The data in the lower word may be literal values (e.g., a character or a 32-bit

integer) or a 32-bit pointer, or they may remain unused when the literal values can be encoded using the tag itself (i.e., `nil`, `false`, `true`). This implementation strategy of encoding pointers and certain literal values in floating-point doubles is efficient because floating-point values—presumably being used extensively in a language dealing with audio and music—do not have to be allocated on the heap, and dereferencing them does not require following a pointer indirection.

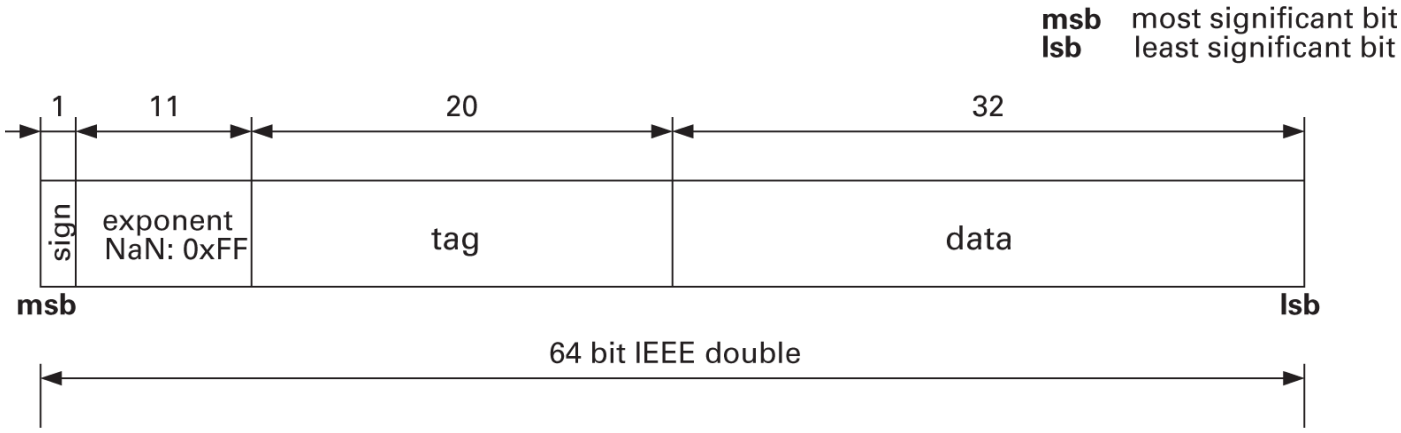


Figure 28.3
32 bit `PyrSlot` memory layout.

Table 28.1
`PyrSlot` tags

Tag	Description
`tagObj`	A pointer to `PyrObject`
`tagInt`	A 32-bit or 64-bit integer
`tagSym`	A pointer to `PyrSymbol`
`tagChar`	Character value
`tagNil`	Literal `nil` (no data)
`tagFalse`	Literal `false` (no data)
`tagTrue`	Literal `true` (no data)
`tagPtr`	A pointer to memory that is not managed by SuperCollider's garbage collector
`tagFloat`	Double floating-point value

However, a problem becomes apparent for 64-bit machines: 64-bit doubles are no longer big enough both to hold a machine address and to store tagging information, so another implementation strategy has been adopted for truly 64-bit address spaces. In the 64-bit version, a `PyrSlot` is a struct consisting of a `long` tag, followed by a `union` to reflect the various data possibilities.

`PyrSlot` members should rarely be manipulated directly, but rather by use of the convenience macros and functions defined in *PyrSlot.h* (e.g., `slotIntVal`,

setSlotVal)–note that some of the preprocessor defines occasionally clash with symbols in system or third-party headers; thus, it is often advisable to include sclang's header files *after* any others.

Each object in SuperCollider includes a header with meta-information concerning the object's class, data format, size, and garbage-collector state, represented by the structure PyrObjectHdr (figure 28.4).

```
struct PyrObjectHdr {
    // garbage collector links
    struct PyrObjectHdr *prev, *next;
    // class pointer
    struct PyrClass* classptr;
    // object size
    int size;

        // indexable object format
    unsigned char obj_format;
    // object size class (power of two)
    unsigned char obj_sizeclass;
    // object flags
    unsigned char obj_flags;
    // garbage collector color
    unsigned char gc_color;

. . .// access and convenience methods
};
```

Figure 28.4
PyrObjectHdr data structure.

The prev and next fields hold object pointers used by the noncopying garbage collector *treadmill* (see section 28.6); classptr points to the object's class (a PyrObject itself), while size holds the number of slots or indexable elements contained in the object. The object's contents are further specified in the obj_format field, whose possible values are listed in table 28.2; obj_sizeclass is the object's power-of-two size class—obtained by a log2Ceil operation on the object's size—and is used by the storage allocator. Further, obj_flags contains special flags concerning the object's state or interpretation; possible values are listed in table 28.3. Finally, gc_color is used by the garbage-collector graph-coloring algorithm.

Table 28.2
Object formats

Format	Object Contents
obj_notindexed	Nonindexed object
obj_slot	PyrSlots
obj_double	64-bit doubles
obj_float	32-bit floats
obj_int32	32-bit integers
obj_int16	16-bit integers
obj_int8	8-bit integers
obj_char	8-bit characters
obj_symbol	Pointers to PyrSymbol

Table 28.3
Object flags

Flag	Description
obj_immutable	Set if object may not be updated
obj_inaccessible	Set if object requires finalization
obj_marked	Used by garbage collector debug sanity check

PyrObject, defined in *PyrObject.h*, establishes the basic structure of any SuperCollider object and is nothing more than an object header followed by a variable number of *slots* representing the object's instance variables. The obj_format field is used to describe the data type of objects contained in Arrays or RawArrays. All objects other than Arrays or RawArrays have obj_format set to obj_notindexed. Arrays can contain only pointers to PyrObject structures, whereas the contents of RawArrays can be symbol pointers or flat scalar data (integers of various sizes, 32-bit floats, and 64-bit doubles), effectively halving the memory requirements for arrays of those data types because the storage management overhead of PyrSlot can be dispensed with (figure 28.5).

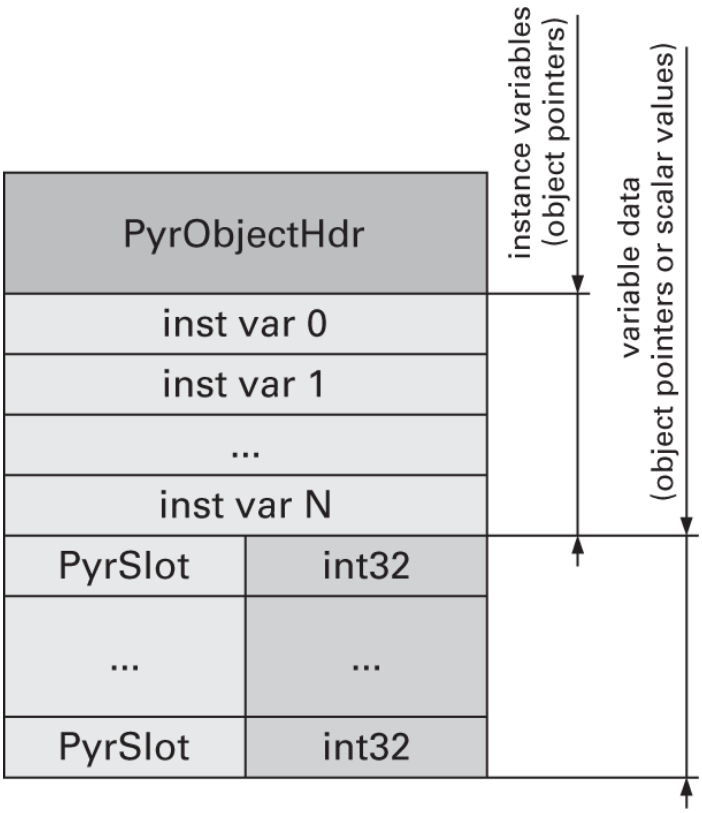


Figure 28.5
Object memory layout.

Symbols—globally unique character strings that can be tested for pointer equality—are represented by a special structure, `PyrSymbol` (figure 28.6), which does not contain an object header and is treated specially by the garbage collector: memory allocated for a symbol is not reclaimed so long as SuperCollider runs. Symbols also point to associated information that is relevant for compilation and interpretation, such as the class object pointer for class name symbols and a selector index for method name symbols.

```
struct PyrSymbol
{
    // symbol name
    char      *name;
    // hash value
    long      hash;
    // special selector index
    short   specialIndex;
    // symbol flags
    uint8   flags;
    // length of symbol name
    uint8   length;
```

```
    union {
                // index in method table or primitive table
                intptr_t index;
                // pointer to class with this name
                struct PyrClass *classobj; name.
                // source code for sym_Filename; used only during com
pilation
                char* source;
    } u;
    // class dependancy (used during compilation)
    struct classdep* classdep;
};
```

Figure 28.6
`PyrSymbol` data structure.

Most of the global information used by compiler and interpreter is stored in the structure `VMGlobals` (figure 28.7). An instance of this structure is held in the global variable `gMainVMGlobals` and is passed to primitives implemented in C. The most important members of this structure are pointers to the runtime stack, the garbage collector, the symbol table, and the current stack and instruction pointers.

```
struct VMGlobals
{
    // global context
    class AllocPool   *allocPool;
    // main thread context
    struct PyrProcess   *process;
    // global symbol table
    class SymbolTable   *symbolTable;
    // garbage collector for this process
    class PyrGC   *gc;
    // class variable array
    PyrObject*   classvars;

    // next byte code is a tail call
    int  tailCall;

    // true when in 'main' thread
    bool  canCallOS;

    // thread context
    struct PyrThread* thread;
    struct PyrMethod* method;
```

```
    struct PyrBlock*  block;
    struct PyrFrame*  frame;
    struct PyrMethod* primitiveMethod;

    // current instruction pointer
    unsigned char* ip;
    // current stack pointer
    PyrSlot*   sp;

    // argument pointer for primitive
    PyrSlot* args;
    // current receiver
    PyrSlot  receiver;
    // interpretation result
    PyrSlot  result;
    // number of args to pop for primitive
    int       numpop;
    // current index into primitive table
    long      primitiveIndex;
    // random number generator state
    RGen*    rgen;
    // handler for unwinding C stack
    jmp_buf  escapeInterpreter;
    // scratch context
    long      execMethod;
// primitive exceptions
    std::map<PyrThread*, std::pair<std::exception_ptr,   PyrMethod*>
> lastExceptions;
};
```

Figure 28.7
`VMGlobals` data structure.

28.4 The Compiler

The purpose of the compiler is to transform expressions given in textual form (e.g., in text files) to byte code for an instruction set that can be interpreted by the interpreter or the virtual machine.

28.4.1 The Compilation Process

The main entry point to the compiler is `compileLibrary`. It initializes the part of the runtime system needed during compilation (`initPassOne`), and in a first pass, it

processes SuperCollider source files found recursively in the main class library directory and the extension directories (`passOne`).

Each class definition found is parsed into a `ClassDependancy` structure that records the class's position in the hierarchy tree, as well as the source file location of its definition (figure 28.8). It is stored in the `classdep` field of the `PyrSymbol` corresponding to the class name. In a first pass, only the class symbols are stored in the `ClassDependancy` because when the list is traversed, classes are not yet hierarchically ordered. Then `buildDepTree` traverses the dependancy tree and records the actual links to the superclass dependancy, further constructing the subclass dependancy list. In `traverseFullDepTree`, the dependancy tree is first flattened into an array of class dependancies (`traverseDepTree`). Finally, the serialized dependancy tree is compiled by `compileDepTree` (for regular classes) or by `compileClassExtensions` (for class extensions).

```
typedef struct classdep
{
   // next link in list
   struct classdep* next;
   // superclass dependency
   struct classdep* superClassDep;
   // subclass list (linked via 'next')
   struct classdep*  subclasses;
   // class name symbol
   PyrSymbol*          className;
   // superclass name symbol
   PyrSymbol*         superClassName;
   // file name symbol
   PyrSymbol*          fileSym;
   // start character position of definition
   int    startPos;
   // end character position of definition
   int    endPos;
   // line number of definition
   int    lineOffset;
} ClassDependancy;
```

Figure 28.8
`ClassDependancy` data structure.

Here, `compileClass` is at the heart of the compiler: the parser generated from the bison grammar (Donnelly and Stallman, 2003) in *Source/LangSource/Bison/lang11d* parses each class according to its source location recorded in the class dependancy and

returns a parse node in the global variable `gRootParseNode` of type `PyrParseNode`. A list of `ParseNode` subclasses, each representing a structural element found in SuperCollider source code, is given in table 28.4. Each `ParseNode` subclass redefines the virtual method compile to emit code according to the structure that it represents.

Table 28.4
Parse node classes

Parse Node Class	Description
`ArgListNode`	Method or function argument list
`AssignNode`	Variable assignment
`BinopCallNode`	Method call (*binary* selector)
`BlockNode`	Function definition
`BlockReturnNode`	Return from function (last expression value)
`CallNode`	Method call
`ClassExtNode`	Class extension
`ClassNode`	Class definition
`CurryArgNode`	*Curried* function arguments
`DropNode`	Drop expression result
`DynDictNode`	Dynamic dictionary definition
`DynListNode`	Dynamic array definition
`LitDictNode`	Literal dictionary definition
`LitListNode`	Literal array definition
`LiteralNode`	Literal value
`MethodNode`	Method definition
`MultiAssignNode MultiAssignVarListNode`	Multiple variable assignment
`PushKeyArgNode`	Keyword argument
`PushLitNode`	Push literal value
`PushNameNode`	Push object by name
`ReturnNode`	Return from method (*caret* return)
`SetterNode`	Method call (*setter* selector)
`SlotNode`	Parse node representing named object
`VarDefNode VarListNode`	Variable definition

The abstract syntax tree returned by the parser entry point `yyparse` is traversed by `compileNodeList`, which in turn calls the compile method for each `ParseNode`. The compiler holds the state in a couple of global variables defined in *PyrParseNode.cpp*, most notably `gCompilingClass` and `gCompilingBlock`, which point to the currently compiling class object and the current block of compiled byte codes, respectively. Byte codes are appended to the current block by the compile methods of the `ParseNodes` in a method or function definition, which end up calling the low-level functions defined in *ByteCodeArray.cpp* that handle byte compilation and memory allocation.

Class compilation is finished by `traverseFullDepTree2`, which first builds the final class tree with `PyrClass` objects (`buildClassTree`), and then indexes it according to the number of classes and the number of methods defined in each class (`indexClassTree`), and finally assigns a unique index to each symbol selector (`setSelectorFlags`)—information that is used for building the method dispatch table in the next step.

Method dispatch in SuperCollider is not based on method selector dictionary lookup; instead, it is realized by indexing into a global dispatch table—the index is held in the class object and the selector symbol, respectively (see figure 28.6). In a simple implementation, the two-dimensional dispatch table, with rows representing classes and columns denoting selectors, would be largely empty because each class defines only a small subset of all method selectors. Here, `buildBigMethodMatrix` builds a compressed table using a row-displacement compression technique (Driesen and Hölzle, 1995), significantly reducing the table's size while retaining the advantages of low-overhead *O(1)* method dispatch.

28.4.2 Byte Codes

The instruction set used in the SuperCollider virtual machine comprises variable-length byte codes. The primary byte code (8-bit) specifying the operation to be executed by the interpreter can be followed by up to four additional secondary bytes of 8 bits each, whose interpretation depends on the primary code.

In the following discussion, we will mention some broad groups of byte codes and their associated functionality. *Push byte codes* push a SuperCollider object onto the stack (e.g., instance and class variables, temporary variables, and pseudovariables such as `self`, the current receiver, and literal values.) *Store byte codes* store an object on top of the stack in an instance variable, class variable, or temporary variable location. *Send byte codes* send a message to a receiver on the stack (along with optional arguments) or execute a primitive function. *Jump byte codes* perform conditional (or unconditional) jumps by modifying the instruction pointer accordingly. Finally, *special byte codes* are in-line implementations that are intrinsic to the interpreter and are generated by the compiler for certain message sends (e.g., `do`, `for`, `forBy`, `if`, `ifNil`, and `switch`).

Table 28.5 lists all the byte codes used by the interpreter, with a short synopsis and the semantics of the corresponding extended byte codes for each. For extended byte codes with one operand, their argument is encoded in the opcode directly (if it fits into 4 bits) by shifting the byte code to the left by 4 and using the lower 4 bits for the operand value. In all other cases, extended byte-code operands are compiled as separate byte codes following the primary one.

Table 28.5
Virtual machine byte codes, listed along with the corresponding constants from `Opcodes.h` where appropriate

		Secondary Byte Code			
Primary Byte Code	Synopsis	1	2	3	4
0 opPushInstVar	push class push instance variable	class index inst var index			
1 opPushTempVar	push temporary variable	temp var level	temp var index		
2 opPushTempZeroVar	push temporary variable from current frame	temp var index			
3 opPushLiteral	push literal selector	literal index			
4 opPushClassVar	push class variable	class var literal index	class var index		
5 opPushSpecialValue	push special class	class name index			
6 opStoreInstVar	store instance variable	inst var index			
7 opStoreTempVar	store temporary variable	temp var level	temp var index		
8 opStoreClassVar	store class variable	class var literal index	class var index		
9 opSendMsg	send message	num args	num key args	selector index	
10 opSendSuper	send `super` message	num args	num key args	selector index	
11 opSendSpecialMsg	send special message	num args	num key args	selector index	
12 opSendSpecialUnaryArithMsg	send unary arithmetic message	selector index			
13 opSendSpecialBinaryArithMsg	send binary arithmetic message	selector index			
14 opSpecialOpcode	special opcode	oppress	push thisProcess		
15		opgMethod	push thisMethod		
		opgFunctionDef	push thisFunctionDef		
		opgFunction	push thisFunction		
		opgThread	push thisThread		
opPushInstVar 16..31	push instance variable 0..15				

		Secondary Byte Code			
Primary Byte Code	Synopsis	1	2	3	4
32	jump if true	jump length (high byte)	jump length (low byte)		
`opPushTempVar` 33..39	push temporary variable levels 1..7	temp var index			
40	push literal constant	index (single byte)			
41	push literal constant	index (byte 1)	index (byte 0)		
42	push literal constant	index (byte 2)	index (byte 1)	index (byte 0)	
43	push literal constant	index (byte 3)	index (byte 2)	index (byte 1)	index (byte 1)
44	push integer constant	single byte			
45	push integer constant	byte 1	byte 0		
46	push integer constant	byte 2	byte 1	byte 0	
47	push integer constant	byte 3	byte 2	byte 1	byte 0
`opPushTempZeroVar` 48..63	push temporary variable 0..15 from current frame				
`opPushLiteral` 64..79	push literal constant 0..15				
`opPushClassVar` 80..95	push class variable	class var index			
`opPushSpecialValue`					
`opsvSelf` 96	push `self`				
`opsvMinusOne` 97	push one and subtract				
`opsvNegOne` 98	push -1				
`opsvZero` 99	push 0				
`opsvOne` 100	push 1				
`opsvTwo` 101	push 2				
`opsvFHalf` 102	push 0.5				
`opsvFNegOne` 103	push -1.0				

`opsvFZero` 104	push 0.0		
`opsvFOne` 105	push 1.0		
`opsvFTwo` 106	push 2.0		
`opsvPlusOne` 107	push one and add		
`opsvTrue` 108	push `true`		
`opsvFalse` 109	push `false`		
`opsvNil` 110	push `nil`		
`opsvInf` 111	push `inf`		
`opStoreInstVar` 112..127	push instance variable 0..15		
`opStoreTempVar` 128..135	store temporary variable levels 0..7	temp var index	
136	push instance variable,send special selector	inst var index	selector index
137	push all arguments, send message	selector index	
138	push all but first argument, send message	selector index	
139	push all arguments, send special selector	selector index	
140	push all but first argument, send special selector	selector index	
141	one argument pushed, push all but first argument, send message	selector index	
142	one argument pushed, push all but first argument, send special selector	selector index	
143	loop and branch byte codes	0..1 `Integer-do` 2..4 `Integer-reverseDo` 5..6 `Integer-for` 7..9 `Integer-forBy` 10 `ArrayedCollection-do` 11..12 `ArrayedCollection-reverseDo` 13..16 `Dictionary-keysValuesArrayDo` 17..18 `Float-do`	

opsvFZero 104	push 0.0	
		19..21 Float-reverseDo 22 method? 23 method?? 24 ifNil 25 ifNotNil 26 ifNotNilPushNil 27 ifNilPushNil 28 switch 29..31 Number-forSeries
opStoreClassVar 144..159	store class variable	class var index
opSendMsg 160..175	send message with 0..15 arguments	selector index
176	return from function (tail call)	
opSendSuperMsg 177..191	send super message with 1..15 arguments	selector index
opSendSpecialMsg 192..207	send special message with 0..15 arguments	selector index
opSendSpecialUnaryArithMsg		
opNeg 208	special unary message negate	
opNot 209	special unary message not	
opIsNil 210	special unary message isNil	
opNotNil 211	special unary message notNil	
212..223	send special unary message with selector index 4..15	
opSendSpecialBinaryArithMsg		
opAdd 224	special binary message +	
opSub 225	special binary message -	
opMul 226	special binary message *	
227..239	send special binary message with selector index 3..15	
opSpecialOpcode		
opcDrop 240	drop value on top of stack	
opcDup 241	duplicate value on top of stack	

opsvFZero 104	push 0.0		
opcFunctionReturn 242	return from function		
opcReturn 243	return from method		
opcReturnSelf 244	return `self`		
opcReturnTrue 245	return `true`		
opcReturnFalse 246	return `false`		
opcReturnNil 247	return `nil`		
opcJumpIfFalse 248	jump if `false`	jump length (high byte)	jump length (low byte)
opcJumpIfFalsePushNil 249	jump if `false` and push `nil`	jump length (high byte)	jump length (low byte)
opcJumpIfFalsePushFalse 250	jump if `false` and push `false`	jump length (high byte)	jump length (low byte)
opcJumpIfTruePushTrue 251	jump if `true` and push `true`	jump length (high byte)	jump length (low byte)
opcJumpFwd 252	jump forward	jump length (high byte)	jump length (low byte)
opcJumpBak 253	jump backward	jump length (high byte)	jump length (low byte)
opcSpecialBinaryOpWithAdverb 254	special binary message with adverb	selector index	
255	return from method (tail call)		

28.4.3 Byte Code Storage

Compile-time information for functions and methods is held in a `PyrBlock` structure (figure 28.9); the actual byte codes are stored in a byte array pointed to by the code field unless the method is in-lined to a special byte code.

```
PyrBlock: PyrObjectHdr
{
    // pointer to PyrMethodRaw
    PyrSlot rawData1;
    // byte codes, nil if inlined
```

```
    PyrSlot code;
    // method selectors, class names, closures table
    PyrSlot selectors;
    // literal constants table
    PyrSlot constants;
    // temporary variable default values
    PyrSlot prototypeFrame;
    // defining block context
    // (nil for methods and toplevel)
    PyrSlot contextDef;
    // arguments to block
    PyrSlot argNames;
    // variables in block
    PyrSlot varNames;
    // source code (for closed functions)
    PyrSlot sourceCode;
};
```

Figure 28.9
`PyrBlock` data structure.

`PyrMethodRaw` is not a proper SuperCollider object; it is stored as a raw machine pointer, and its purpose is to record information regarding the type of a `PyrBlock` and the size and structure of its runtime activation frames (figure 28.10). The `specialIndex` field contains an integer index used in the special handling of runtime behavior, such as primitive invocation and accessing class and instance variables. Another type of optimization is realized by `methType`, denoting various special method types such as returning the receiver and returning and assigning instance variables (table 28.6).

```
struct PyrMethodRaw {
#ifdef PYR_SLOTS_GENERIC
 long padding; // used for the tag in the generic pyrslot   impleme
ntation
#endif
    unsigned short unused1;
    // special method index
    unsigned short specialIndex;
    // method type
    unsigned short methType;
    // prototype frame size
    unsigned short frameSize;

#ifdef PYR_SLOTS_GENERIC
    long padding2; // used for the tag in generic pyrslot   implemen
```

```
tation, second slot
#endif

    unsigned char unused2;

    // number of arguments
    unsigned char numargs;
    // 1 if has variable number of arguments
    unsigned char varargs;
    // number of keyword and variable defaults
    unsigned char numvars;
    // number of temporary (local) variables
    unsigned char numtemps;
    // true when frame needs to be heap-allocated
    unsigned char needsHeapContext;
    // used for debugging
    unsigned char popSize;
    // numargs + varargs
    unsigned char posargs;
};
```

Figure 28.10
`PyrMethodRaw` data structure.

Table 28.6
Method type enumeration values

`methNormal`	Normal method invocation
`methReturnSelf`	Return the receiver
`methReturnLiteral`	Return a literal value
`methReturnArg`	Return a method argument
`methReturnInstVar`	Return an instance variable
`methAssignInstVar`	Assign an instance variable
`methReturnClassVar`	Return a class variable
`methAssignClassVar`	Assign a class variable
`methRedirect`	Send a different selector to `self`
`methRedirectSuper`	Send a different selector to `self`, start `lookup` in the superclass
`methForwardInstVar`	Forward message send to an instance variable
`methForwardClassVar`	Forward message send to a class variable
`methPrimitive`	Execute a primitive
`methBlock`	Denotes a function block

Methods are a special case of ordinary code blocks, and they store some additional information such as class, method name, and source code location (figure 28.11).

```
struct PyrMethod: public PyrBlock
{
    PyrSlot ownerclass;
    PyrSlot name;
    PyrSlot primitiveName;
    PyrSlot filenameSym;
    PyrSlot charPos;
};
```

Figure 28.11
`PyrMethod` data structure.

28.5 The Interpreter

The interpreter's task is to execute—or *dispatch*—byte codes in a loop once they have been compiled into method and function definitions by the compiler. It maintains an instruction pointer (IP) indicating the currently executed byte code; after execution of a primary byte code, the IP may point to a byte code in the same block or, in the case of function and method calls, to a totally different code block.

Arguments are passed through a global stack which is held in the `VMGlobals` structure described above. The interpreter maintains a pointer to the current top of the stack and increases or decreases the pointer as arguments are pushed and popped. (The stack grows toward higher addresses.) The receiver of a message is always pushed first, followed by the fixed-position arguments. Variable argument passing depends on the message being sent; normal message sends and function evaluation collect variable arguments in an `Array` that is passed as the last argument on the stack, while calls to primitives push variable arguments onto the stack directly (figure 28.12).

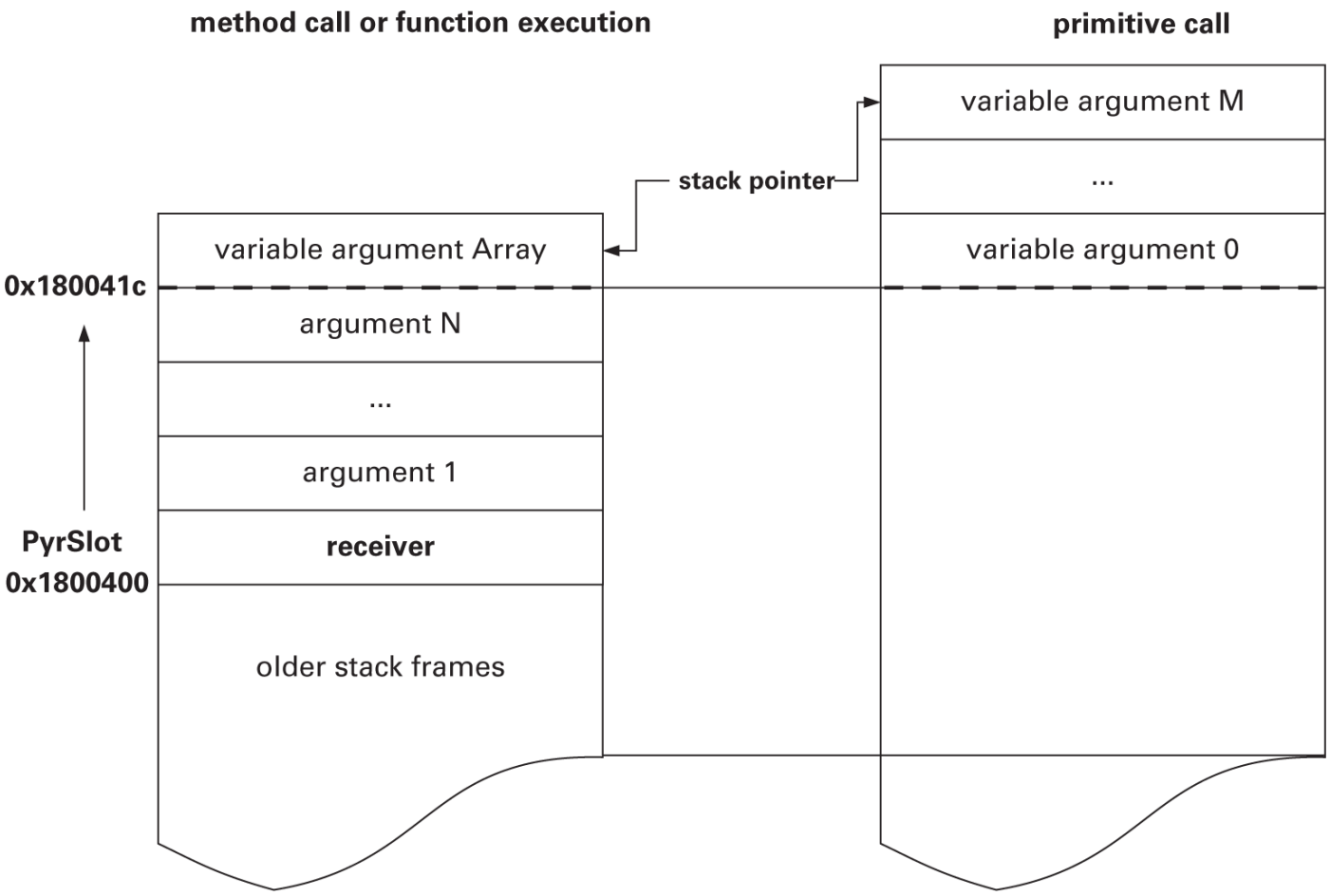


Figure 28.12
Stack layout for message sends and primitive calls.

When a function or method is executed, all the runtime information that is needed for an activation frame is kept in a `PyrFrame` structure: the calling context, the closure context (for function activations), the enclosing method (*home*) context, the activation frame's instruction pointer, and storage for temporary variables (figure 28.13).

```
struct PyrFrame: public PyrObjectHdr

{
    // defining method
    PyrSlot method;
    // calling context
    PyrSlot caller;
    // closure context
    PyrSlot context;
    // method context
    PyrSlot homeContext;
    // instruction pointer
    PyrSlot ip;
```

```
    // temporary variable storage
    PyrSlot vars[1];
};
```

Figure 28.13
`PyrFrame` data structure.

The `executeMethod` and `blockValue` functions are the principal means of executing methods and functions (*blocks*), respectively, and are mostly identical in their operation, except for the closure context used by block activations, which captures a function's lexical environment. Here is a rundown of `executeMethod`'s operation:

0. Receiver and arguments pushed onto stack, method already looked up via selector index by `sendMessage`.
1. Create new frame, record calling context, and home context.
2. Save caller IP.
3. Update interpreter state: SP, IP, current frame, and current code block.
4. Process default and keyword arguments.
5. Return to interpreter and resume at new IP.

When returning from a method, the interpreter calls the `returnFromMethod` function (except in some specially optimized cases). Its purpose is to unwind the call chain, represented as a list of activation frames linked by the caller field (figure 28.14), and to resume execution at the method context's calling context:

1. If the current frame's `homeContext` exists (i.e., it hasn't been returned from yet), unwind the call chain to `homeContext`'s caller, or else unwind to the top-level interpreter frame.
2. Set interpreter IP to `returnContext`'s saved IP.
3. Return to interpreter and resume at new IP.

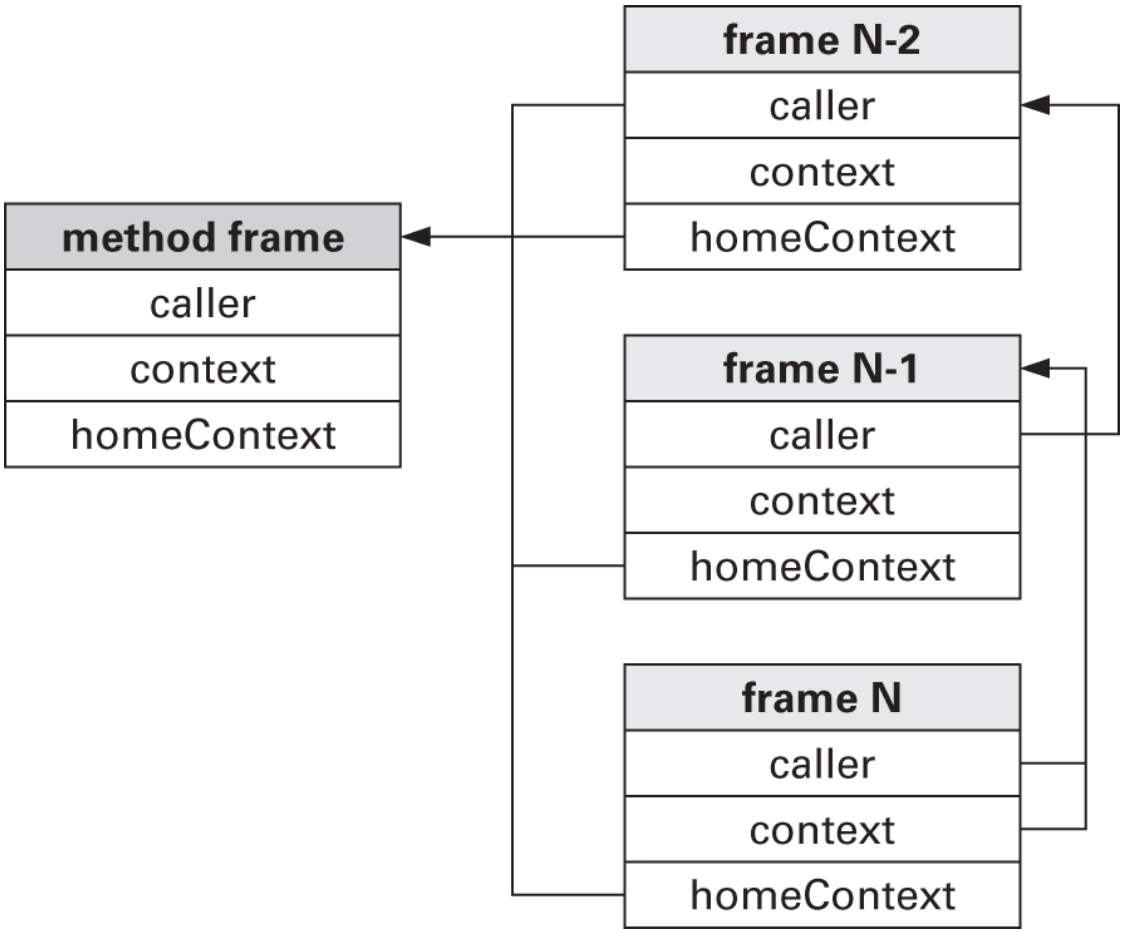


Figure 28.14
Activation frame call chain.

The interpreter's main entry point is `Interpret` (defined in *PyrInterpreter3.cpp*), which expects the interpreter state to be correctly initialized and the stack and instruction pointer to be set up for execution. The stack and instruction pointer are first copied to function local storage (in the hope that the compiler puts them into registers), and then the byte code pointed to by the IP is dispatched in a big `switch` statement.

A higher-level interface to the interpreter is `runInterpreter`, which takes care of interpreter initialization and cleanup and sends a selector symbol to the receiver that is expected to be pushed on the stack, followed by the message arguments:

```
void runInterpreter(VMGlobals *g, PyrSymbol *selector,   int numArg
sPushed);
```

Figure 28.15 shows the most important structures and classes used by the interpreter and their relationships. In the C code representation, all structs representing language objects have `PyrSlot` members lacking any type information. In the class diagram, the following approach has been chosen instead: attribute types are named according to the

corresponding language entity unless the class has a different name in the C code (UML comments signify the correspondence). For clarity, the prefix “Pyr” has been omitted in the class diagram.

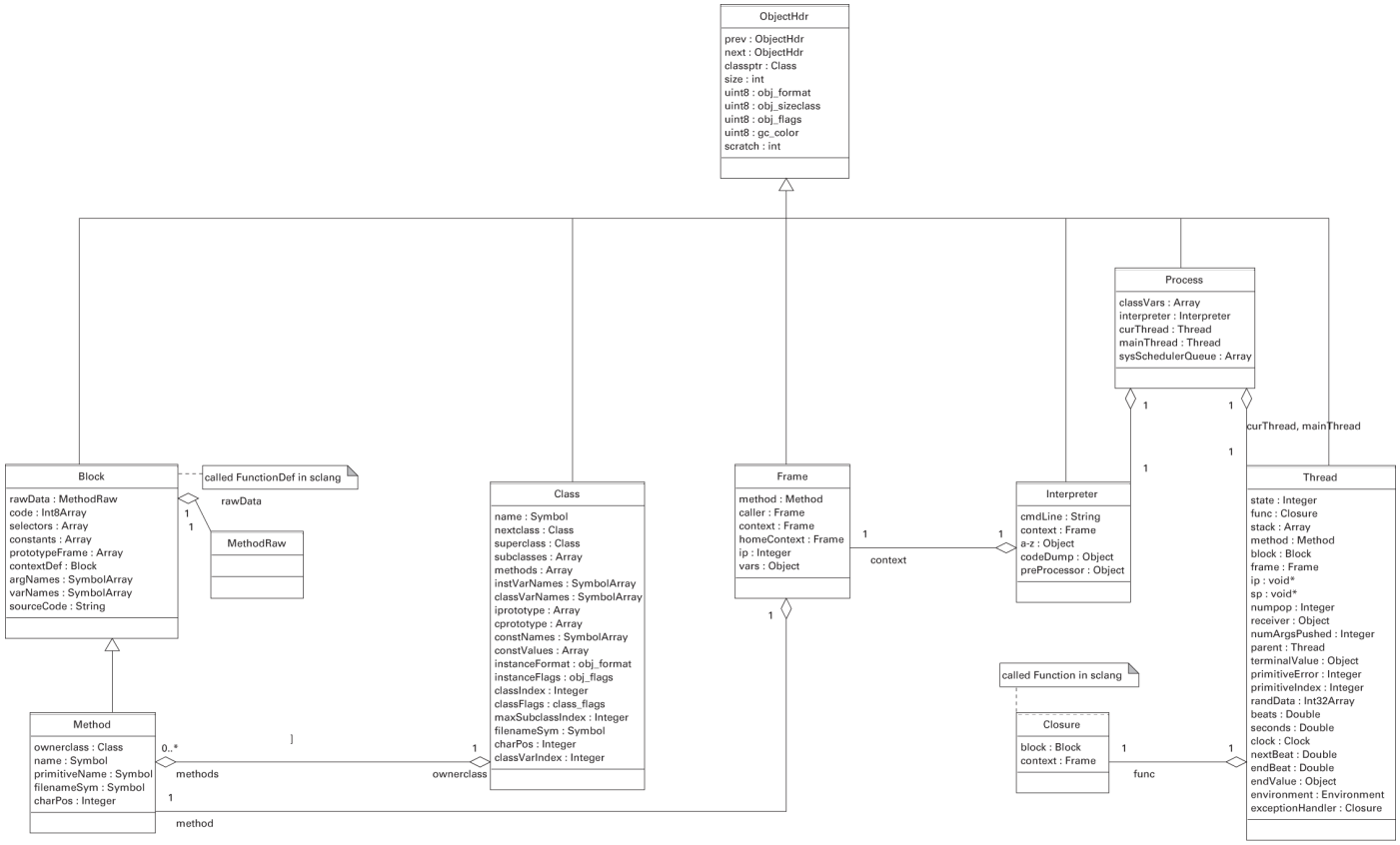


Figure 28.15
Most important `Interpreter` implementation entities (some details omitted).

28.6 The Garbage Collector

SuperCollider uses an incremental noncopying garbage collector (Wilson and Johnstone, 1993; Wilson et al., 1995) that can guarantee an upper bound for the amount of collection proportional to the amount of memory allocated per time interval, which is necessary in real-time applications that require an upper bound on GC pauses, such as audio synthesis and algorithmic composition.

A small and bounded amount of garbage collection is done whenever a storage allocation request is handled. Objects are not copied during collection but are kept in circular lists—the *treadmills*—that are incrementally traversed by the collector. The `PyrObjectHdr` fields `prev` and `next` link an object into the treadmill. Incremental collection is seen as a graph-coloring problem: the `gc_color` field of `PyrObjectHdr` contains 1 of the 3 colors *white, gray, black* or the special tag *free*, denoting the state of garbage collection that a given object is currently in:

- *White* objects have not yet been considered by the incremental traversal.
- *Gray* objects are known to be reachable, but their contained pointers have not yet been fully inspected.
- *Black* objects are known to be reachable, and all their fields have been scanned and colored *gray*.
- *Free* objects are free to be reused by subsequent allocations.

As the collector colors objects, they are moved to a linked list corresponding to the respective color; incremental collection finishes when the *gray* list does not contain any objects to scan. At this point, the *white* list contains objects that can be freed—they are appended to the *free* list in an *O(1)* operation—and the *black* list contains all live objects.

Because the mutator—the running program—might change the object graph between incremental collections, those updates have to be communicated to the garbage collector. In particular, the following invariant must be maintained: An object known to be reachable (*black* object) may point only to other *black* objects or ones that are currently under consideration by the garbage collector (*gray* objects). Storing a *white* object directly into a *black* object and discarding all other references would result in a dangling pointer because the *white* object's storage space will be reclaimed at the end of a complete garbage-collection pass.

This invariant is maintained by a *write barrier* (i.e., the garbage collector is notified whenever a pointer to a *white* object is stored in a *black* object). The method `GCWrite`, defined in *GC.h*, implements the write barrier by coloring the child *gray* when it is currently *white* and the parent is *black*:

```
void GCWrite(PyrObjectHdr* parent, PyrObjectHdr* child);
```

Variants of this method provide an optimized write barrier when the parent is known to be *black* (`GCWriteBlack`) and when the child is known to be white (`GCWriteNew`), such as when it was allocated by the preceding call to the garbage collector (newly allocated objects are colored *white*).

28.7 Writing Primitives

Sometimes it is necessary to add methods implemented in C to the SuperCollider language in order to interface to third-party libraries, access operating system services, or simply implement an operation as efficiently as possible. SuperCollider already contains a large number of primitive method implementations, which are an invaluable resource in writing your own primitives.

At the time of this writing, the SuperCollider language (as opposed to the synthesis server) does not support primitives to be loaded from plug-in modules or dynamic libraries. This means that a primitive implementation has to be included as a module in SuperCollider's source code and be added to the build system; the details of how to do this are platform-dependent and will not be explained here. Because of the "intrusiveness" of primitive implementations, the first question to answer when trying to implement a particular primitive (or set of primitives) is whether it is needed. Often the increase in performance is negligible to an implementation in pure sclang, or there are alternative solutions that might be better suited to the problem at hand, such as communicating with external processes through *OpenSoundControl* (see chapter 4) or reading external command output (see `systemCmd`, `unixCmd`, and `Pipe`).

Primitives are implemented by registering a primitive *handler* function with the SuperCollider runtime system. Here, `definePrimitive` is the main means of registering new primitives with a fixed or variable number of arguments:

```
typedef int (*PrimitiveHandler)(struct VMGlobals* g,   int numArgsP
ushed);
int definePrimitive(
   int base, int index, const char* name,
   PrimitiveHandler handler,
   int numArgs, int varArgs
);
```

Here, `base` is a unique base index for a primitive module or collection of primitives; its initial value should be obtained by a call to `nextPrimitiveIndex`; `index` is an index for a particular primitive in the same module and should be incremented for each primitive registered; `name` is the identifier that is introduced to the runtime system and is also used by the compiler when encountering a primitive reference in the source code (the convention is to have primitive names begin with an underscore _ and a capital character); `handler` is a C function pointer, with the signature of `PrimitiveHandler` above being called by the interpreter upon primitive execution. Finally, `numArgs` specifies the number of fixed arguments that a primitive is expected to be called with (including the receiver self), and `varArgs` should be true (1) when the primitive expects a variable number of arguments after the fixed-positional arguments.

At a minimum, the following header files need to be included in each source module with primitive definitions—more might be needed for additional functionality:

```
#include "GC.h"
#include "PyrKernel.h"
#include "PyrPrimitive.h"
```

Primitive definitions usually start by extracting arguments from the stack into a C representation, returning an appropriate error code when invalid arguments are encountered. Extracted arguments are then passed to a library or operating system function that carries out the desired low-level operation. The result is converted back to a SuperCollider object and stored on the stack before control is passed to the interpreter by returning from the primitive.

The receiver (`self`) of the current primitive invocation can be found on the stack below the pushed arguments (i.e., at `g->sp-numArgsPushed + 1`), and the argument slots are at the subsequent `numArgsPushed` stack locations. The header files PyrSlot.h, PyrSlot64.h, and PyrSlot32.h (the last two for 64- and 32-bit systems respectively) define a number of utility functions for extracting values from `PyrSlot` values and converting them to a representation compatible with C, as well as modifying slot tags and data (table 28.7). The returned value should replace the first argument pushed on the stack (i.e., the receiver), where the interpreter expects it after the primitive returns.

Table 28.7
`PyrSlot` access functions

Modifying Slots (Tags and Data)	
`SetInt`	Set slot to 32-bit integer value.
`SetObject`	Set slot to object pointer.
`SetSymbol`	Set slot to symbol pointer.
`SetChar`	Set slot to character.
`SetPtr`	Set slot to raw pointer.
`SetObjectOrNil`	Set slot to object pointer (or `nil` for `NULL`).
`SetTrue`	Set slot to `true`.
`SetFalse`	Set slot to `false`.
`SetBool`	Set slot to Boolean value.
`SetNil`	Set slot to `nil`.
`SetFloat`	Set slot to 64-bit floating-point value.
`SetRaw, SetRawChar, SetTagRaw`	Raw setter functions (overloaded) for various types without typecheck.
Querying Slots	
`IsNil NotNil`	Whether slot is `nil`
`IsFalse IsTrue`	Whether slot is `true` or `false`
`IsInt NotInt`	Whether slot is an integer value
`IsFloat NotFloat`	Whether slot is a floating-point value
`IsObj NotObj`	Whether slot is an object pointer
`IsSym NotSym`	Whether slot is a symbol pointer
`IsPtr NotPtr`	Whether slot is a raw pointer
`IsChar NotChar`	Whether slot is a char

Modifying Slots (Tags and Data)	
`SlotEq`	Whether two slots are equal (i.e., with the same tag and data)
Accessing Slot Data	
`int slotIntVal(PyrSlot* slot, int* value);`	Get integer value Return error if `slot` is not a number
`int slotFloatVal(PyrSlot* slot, float* value);`	Get float value Return error if `slot` is not a number
`int slotDoubleVal(PyrSlot* slot, double* value);`	Get double value Return error if `slot` is not a number
`int slotStrVal(PyrSlot* slot, char* str, int maxlen);`	Get `NULL`-terminated string value Return error if `slot` is not a `String` or `Symbol`
`int slotPStrVal(PyrSlot* slot, unsigned char* str);`	Get string value with byte count prefix (*Pascal string*) Return error if `slot` is not a `String` or `Symbol`
`int slotSymbolVal(PyrSlot* slot, PyrSymbol **dstptr);`	Get symbol pointer Return error if `slot` is not a `Symbol`
`inline void slotCopy(PyrSlot* dst, const PyrSlot* src)inline void slotCopy(PyrSlot* dst, PyrSlot* src, int num);`	Copy `src` to `dst`, or `num` slots from `src` to `dst`

Let's take a look at a simple example of primitive implementation of the `atan2` *C* math library function in figure 28.16. (The actual `atan2` operation in sclang is implemented within the `prSpecialBinaryArithMsg` primitive, which handles a range of operations.)

```
#include <math.h>
#include "GC.h"
#include "PyrKernel.h"
#include "PyrPrimitive.h"

// Primitive implementation of atan2,
// calling the function from libm.
static int prAtan2(struct VMGlobals *g, int numArgsPushed)
{
    // Pointer to arguments
    PyrSlot *args = g->sp - numArgsPushed + 1;
    // Pointer to receiver (self)
    PyrSlot* self = args + 0;
    // Pointer to argument
    PyrSlot* arg = args + 1;

    double x, y;
```

```
    int err;

    // Get receiver value
    err = slotDoubleVal(self, &x);
    // Signal error for invalid input type
    if (err != errNone) return err;

    // Get argument value
    err = slotDoubleVal(arg, &y);
    // Signal error for invalid input type
    if (err != errNone) return err;

    // Compute result
    double result = atan2(x, y);

    // Set top of stack to return value
    SetFloat(self, result);

    // Signal success
    return errNone;
}

// Call this function during initialization,
// e.g. from initPrimitives() in PyrPrimitive.cpp
void initMyPrimitives()
{
    // Initialize primitive indices
    int base = nextPrimitiveIndex(), index = 0;

    // Define primitive with two arguments (self, operand)
    definePrimitive(base, index++, "_MyFloatAtan2", prAtan2, 2, 0);
    //. . .define more primitives here. . .
}
```

Figure 28.16
Implementation of an example primitive for `atan2`.

First, the arguments are extracted with the function `slotDoubleVal`, which returns an error when the `slot` argument is not a number object. Then the return value pointed to by the first pushed argument is set to the result of the library call before returning to the interpreter.

Primitives can get much more complicated than that—the primitive implementations in *`Source/Lang/LangPrimSource`* provide an excellent source of example code. The

Writing Primitives Help file provides more detail on this.

The primitives implemented in the Qt-based graphical user interface implementation use a macro, `QC_LANG_PRIMITIVE(NAME, ARGC, RECEIVER, ARGS, GLOBAL)`, to automate primitive definition and simplify stack, argument, and receiver management.

Special care must be taken to handle garbage collector management correctly when writing primitives that create new objects. In brief, before calling any function that might allocate a new object (like `newPyrObject`), you must make sure that these criteria are fulfilled:

- All objects previously created must be reachable, which means that they must exist on the `g->sp` stack (taking care not to overwrite any objects which will be needed later); in a lang-side variable or class variable; or in a slot of another object that fulfills these criteria.
- If any object (`child`) was put inside a slot of another object (`parent`), you must call `g->gc->GCWrite(parent, child)` afterward unless you know that the `parent` is still white (unexamined), or `GCWriteNew` if you also know that the `child` is white; and set `parent->size` to the correct value.

Again, the *Writing Primitives* Help file provides more detail and examples about this important topic.

28.8 Source File Overview

Table 28.8 gives a short overview of the most important source and header files of the core language implementation.

Table 28.8
Language implementation headers and sources

File	Description
GC.h GC.cpp	Garbage collector implementation
PyrInterpreter.h	Interpreter interface functions
PyrInterpreter3.cpp	Virtual machine implementation
PyrKernel.h	Core object structures
PyrLexer.h PyrLexer.cpp	Lexer implementation and parser interface
PyrMessage.h PyrMessage.cpp	Message send interface, method execution, and return
PyrObject.h	Object header and core object structure definitions
PyrObject.cpp	Class tree and method table construction, basic constructors, and primitives
PyrParseNode.h PyrParseNode.cpp	Parse node structure definitions, compilation function implementations
PyrPrimitive.h	Primitive interface
PyrSlot.h PyrSlot32.h PyrSlot64.h	`PyrSlot` definition and interface functions (32- and 64-bit implementations where appropriate)

File	Description
PyrSymbol.h	Symbol structure definition
PyrSymbolTable.h *PyrSymbolTable.cpp*	Symbol table interface
SC_LanguageClient.h *SC_LanguageClient.cpp*	Abstract interpreter interface
SC_LibraryConfig.cpp	Library configuration file handling
SC_TerminalClient.h *SC_TerminalClient.cpp*	Command line interpreter interface
VMGlobals.h	VMGlobals structure definition

References

Budd, T. 1987. *A Little Smalltalk*. Reading, MA: Addison-Wesley.

Donnelly, C., and R. M. Stallman. 2003. *Bison Manual: Using the YACCCompatible Parser Generator*. Boston: GNU Press.

Driesen, K., and U. Hölzle. 1995. "Minimizing Row Displacement Dispatch Tables." In *OOPSLA '95: Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications,* pp. 141–155. New York: ACM Press.

Goldberg, A., and D. Robson. 1983. *Smalltalk80: The Language and Its Implementation.* Reading, MA: Addison-Wesley.

IEEE Standards Board. 1985. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/ IEEE Std, 754–1985.

Ingalls, D., T. Kaehler, J. Maloney, S. Wallace, and A. Kay. 1997. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself." *ACM SIGPLAN Notices,* 32(10).

McCartney, J. 1996. "SuperCollider: A New Real Time Synthesis Language." In *Proceedings of the 1996 International Computer Music Conference,* Hong Kong, pp. 257–258.

Wilson, P. R., and M. S. Johnstone. 1993. "Real-Time Non-copying Garbage Collection." In *ACM OOPSLA Workshop on Memory Management and Garbage Collection.* New York: ACM Press.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles. 1995. "Dynamic Storage Allocation: A Survey and Critical Review." In *Proceedings of the International Workshop on Memory Management,* Kinross, UK, pp. 1–116.

29 Writing Unit Generator Plug-ins

Dan Stowell and Christof Ressi

Writing a unit generator (UGen) for SuperCollider 3 can be extremely useful, allowing the addition of new audio generation and processing capabilities to the synthesis server. The bulk of the work is C++ programming, but the application programming interface (API) is essentially quite simple—so even if you have relatively little experience with C/C++, you can start to create UGens based on existing examples.

You're probably already familiar with UGens from previous chapters. Before creating new UGens of your own, though, let's first consider what a UGen really is, from the plug-in programmer's point of view.

29.1 What Is a UGen, Really?

A UGen is a component of the synthesis server, defined in a plug-in, which can receive a number of floating-point data inputs (audio- or control-rate signals or constant values) and produce a number of floating-point data outputs, as well as side effects such as writing to the post window, accessing a buffer, or sending a message over a network. The server can incorporate the UGen into a synthesis graph, passing data from one UGen to another.

When using the SC language, we need to have available a representation of each UGen which provides information about its inputs and outputs (the number, type, etc.). These representations allow us to define synthesis graphs in SC language (`SynthDefs`). Therefore, each UGen also comes with an SC class; these classes are always derived from a base class, appropriately called `UGen`.

So to create a new UGen, you need to create both the plug-in for the server and the class file for the language client.

29.2 An Aside: Pseudo UGens

Before we create a "real" UGen, we'll look at something simpler. A *pseudo UGen* is an SC class that "behaves like" a UGen from the user's point of view but doesn't involve any new plug-in code. Instead, it just encapsulates some useful arrangement of existing units. Let's create an example: a simple reverb effect, as in figure 29.1.

```
Reverb1 {
    *ar {| in |
        var out = in;
        out = AllpassN.ar(out, 0.05, 0.05.rand, 1);
        ^out;
    }
}
```

Figure 29.1
Reverb1.

This isn't a very impressive reverb yet, but we'll improve it later.

As you can see, this is a class like any other, with a single class method. The `*ar` method name is not special; in fact, you could use any method name (including `*new`). We are free to use the full power of SC language, including constructs such as `0.05.rand`, to choose a random delay time for our effect. The only real requirement for a pseudo UGen is that the method returns something that can be embedded in a synth graph. In our simple example, what is returned is an `AllpassN` applied to the input.

Copy the above code into a new file and save it as something like Reverb1.sc in your SCClassLibrary or Extensions folder; then recompile. You'll now be able to use `Reverb1.ar` within your `SynthDefs`, just as if it were a real UGen. Let's test this in figure 29.2

```
s.boot;
(
x = {
    var freq, son, out;
    // Chirps at arbitrary moments
    freq = EnvGen.ar(Env.perc(0, 0.1, 10000), Dust.ar(1));
    son = SinOsc.ar(freq, 0, 0.1);
    // We apply reverb to the left and right channels separately
    out = {Reverb1.ar(son, cutoff: 2500)}.dup;
}.play(s);
)
x.free;
```

Figure 29.2
`Reverb1` example.

You may also wish to create a Help file for `Reverb1`. This is not too difficult: Type "Reverb1" and look up the documentation via the Help menu. Since the Help file does

not exist yet, the Help browser will automatically create a Help template that you can copy and modify. More about Help files will follow soon.

To make the reverb sound more like a reverb, we now modify it to perform six similar all-pass delays in a row, and we also add some LPF units in the chain to create a nice frequency roll-off. We also add parameters, as in figure 29.3.

```
Reverb1 {
    *ar {| in, wet = 0.3, cutoff = 3000|
        var out = in;
        6.do{out = LPF.ar(AllpassN.ar(out, 0.05, 0.05.rand, 1),   c
utoff)};
        ^(out * wet) + (in * (1-wet));
    }
}
```

Figure 29.3
`Reverb1` additions.

This is on the way to becoming a useful reverb unit without having created a real plug-in at all.

This approach has definite limitations. Of course, it is confined to processes that can be expressed as a combination of existing units; it can't create new types of processing or new types of server behavior. It may also be less efficient than an equivalent UGen because it creates a small subgraph of units that pass data to each other and must maintain their own internal states separately.

Now let's consider what is involved in creating a real UGen.

29.3 Steps Involved in Creating a UGen

1. Consider exactly what functionality you want to encapsulate into a single unit. An entire 808-drum machine, or just the cymbal sound? Smaller components are typically better because they can be combined in many ways within a `SynthDef`. Efficiency should also be a consideration.

2. Write the Help file. Really—it's a good idea to do this before you start coding, even if you don't plan to release the UGen publicly.

As well as being a good place to keep the example code which you can use while developing and testing the UGen, it forces you to think clearly about the input and output and how the UGen will be used in practice, thus weeding out any conceptual errors.

A Help file is also a good reminder of what the UGen does—don't underestimate the difficulties of returning to your own code, months or years later and trying to decipher

your original intentions!

Help files are written in the SCDoc format and have the .schelp extension. See the *WritingHelp* file for a short guide on how to write Help files and the *SCDoc Syntax* file for the documentation of the precise markup language. As we have already shown in the previous section, you can simply start from the Help template created by the Help browser. Of course, during development the Help file doesn't need to be particularly beautiful.

3. Write the class file. You don't need to do this before starting on the C++ code, but it's a relatively simple step. Existing class files (e.g., for `SinOsc`, `LPF`, `Pitch`, `Dwhite`) can be helpful as templates. More on this shortly.

4. Write the plug-in code. The programming interface is straightforward, and again existing plug-in code can be a helpful reference. All UGens are written as plug-ins—including the core UGens—so there are lots of code examples available. Note, however, that the core plug-ins are written in an older style of C++ and may use various macros (e.g., those in *include/plugin_interface/Unroll.h*) that are not necessary for modern compilers and can make the code hard to read.

We now consider writing the class file and writing the plug-in code.

29.4 Writing the Class File

A class file for a UGen is much like any other SC class, with the following conditions:

It must be a subclass of `UGen` so that methods defined in the UGen class can be used when the language builds the SynthDef (synth graph definition).

The name of the class must match the name used in the plug-in code—the class name is used to tell the server which UGen to instantiate.

It must implement the appropriate class methods for the rates at which it can run (e.g., `*ar`, `*kr`, and/or `*ir`). The class methods must call the `multiNew` method (defined in the main `UGen` class), which processes the arguments and adds the `UGen` correctly to the `SynthDef` that is being built.

The class file does not have any direct connection with the C++ plug-in code—after all, it's the server that uses the plug-in code, while the class file is for the language client.

Let's look at a well-known example, in figure 29.4.

```
SinOsc: UGen {
   *ar {
        arg freq = 440.0, phase = 0.0, mul = 1.0, add = 0.0;
        ^this.multiNew('audio', freq, phase).madd(mul, add)
   }
```

```
    *kr {
        arg freq = 440.0, phase = 0.0, mul = 1.0, add = 0.0;
        ^this.multiNew('control', freq, phase).madd(mul, add)
    }
}
```

Figure 29.4
`SinOsc` example.

As you can see, `SinOsc` is a subclass of UGen and implements two class methods. Both of these methods call `multiNew` and return the result, which is one or more instances of the UGen that we are interested in.

The first argument to `multiNew` is a symbol to indicate the rate at which the particular UGen instance will be operating: this could be "audio," "control," "scalar," or "demand." The remaining arguments are those that will actually be passed to the C++ plug-in—here, `freq` and `phase`. If any of these arguments are arrays, `multiNew` performs multichannel expansion, creating a separate unit to handle each channel. Indeed, this is why the method is called `multiNew`.

Often, you would simply forward the arguments of your `*ar` or `*kr` class methods to `multiNew`, but sometimes you might need to transform or validate them in certain ways. We will see an example in section 29.6.2, where we try to pass `Arrays` into UGens. After multichannel expansion, *multiNew* eventually passes the expanded argument lists to the *init* method on each UGen instance. The default implementation of *init* simply assigns the argument array to the *inputs* member. Often, this is exactly what you want, but occasionally you need to override the *init* method to add custom code. We will see some examples in section 29.6.1, when we talk about UGens with multiple outputs.

Note that in our `SinOsc` example, there are also `mul` and `add` arguments that are not being passed to `multiNew`; instead, they call the `madd` method on the resulting UGens. In the past, this was a necessary optimization and it has been a convention to add `mul` and add arguments to many UGens as the final two arguments. Nowadays, this practice is obsolete and these `mul` and `add` arguments are only kept for backward compatibility. Don't use them in new UGens! The code responsible for building the SynthDef applies several optimization passes to the resulting UGen graph. Among others, it recognizes and simplifies common mathematical expressions. For example, an expression like *x* * *y* + *z* is replaced by a single `MulAdd` UGen. It can even do constant folding and eliminate dead code, so that *x* * *(2* * *4)* + *(1* + *3)* becomes *x* * *8* + *4* and `x * 1 + 0` is reduced to *x*. This means that there is no reason to use `mul` and `add` anymore; users can just write ordinary mathematical expressions: `SinOsc.ar(440, mul: 4, add: 2)` yields the exact same result as `SinOsc.ar(440) * 4 + 2`.

Let's start to draft the class file for a UGen that we can implement. We'll create a basic "flanger," which takes some input and then adds an effect controlled by rate and depth parameters (see figure 29.5).

```
Flanger: UGen {
    *ar {
        arg in, rate = 0.5, depth = 1.0;
        ^this.multiNew('audio', in, rate, depth)
    }
    *kr {
        arg in, rate = 0.5, depth = 1.0;
        ^this.multiNew('control', in, rate, depth)
    }
}
```

Figure 29.5
Flanger.

Save this as *Flanger.sc* in your extensions directory. If you recompile, you'll find that this is sufficient to allow you to use `Flanger.ar` or `Flanger.kr` in `SynthDefs`, which the SuperCollider language will happily compile—but of course, those `SynthDefs` won't run yet because we haven't created anything to tell the server how to produce the `Flanger` effect.

29.4.1 Checking the Rates of Your Inputs

Because SuperCollider supports different signal rates, it is useful to add a bit of "sanity checking" to your `UGen` class to ensure that the user doesn't try to connect things in a way that doesn't make sense: for example, plugging an audio-rate value into a scalar-rate input.

The UGen class provides a `checkInputs` method, which you can override to perform any appropriate checks. When the `SynthDef` graph is built, each UGen's `checkInputs` method will be called. The default method defined in UGen simply passes through to `checkValidInputs`, which checks that each of the inputs is really something that can be plugged into a synth graph (and not some purely client-side object such as, say, a `Window` or a `Task`).

The `BufWr` UGen is an example which implements its own rate checking. Let's look at what the class does (see figure 29.6).

```
checkInputs {
    if (rate == 'audio' and: {inputs.at(1).rate != 'audio'}, {
        ^("phase input is not audio rate:" + inputs.at(1) +   input
```

```
    s.at(1). rate);
        });
        ^this.checkValidInputs
    }
```

Figure 29.6
`checkInputs`.

If `BufWr` is used to write audio-rate data to a buffer, then the input specifying the phase (i.e., the position at which data is written) must also be audio-rate—there's no natural way to convert a control-rate input to an audio-rate buffer index. Therefore, the class overrides the `checkInputs` method to test explicitly for this. The rate variable is the rate of the unit under consideration (a symbol, just like the first argument to `multiNew`). The inputs variable is an array of the unit's inputs, each of which will be a UGen and thus will also have a rate variable. So the method compares the present unit's rate against its first input's rate. It simply returns a string if there's a problem (returning anything other than *nil* is a sign of an error found while checking input). If everything is OK, it passes through to the default `checkValidInputs` method—if you implement your own method checking, don't forget to do this!

Many UGens produce output at the same rate as their first input—for example, filters such as `LPF` or `HPF`. If you look at their class definition (or their superclass, in the case of `LPF` and `HPF`—an abstract class called `Filter`), you'll see that they call a convenience method for this common case called `checkSameRateAsFirstInput`. Observe the result of these checks in figure 29.7.

```
s.boot;
x = {LPF.ar(WhiteNoise.kr)}.play(s); // Error
x = {LPF.ar(WhiteNoise.ar)}.play(s); // OK
x.free;
x = {LPF.kr(WhiteNoise.ar)}.play(s); // Error
x = {LPF.kr(WhiteNoise.kr)}.play(s); // OK
x.free;
```

Figure 29.7
`checkSameRateAsFirstInput`.

What happens if you don't add rate checking to your UGens? Often, it makes little difference, but ignoring rate checking can sometimes lead to unusual errors that are hard to trace. For example, a UGen that expects control-rate input is relatively safe because it expects less input data than an audio-rate UGen—so if it is given audio-rate data, it simply ignores most of it. But in the reverse case, a UGen that expects audio-rate data

but is given only control-rate data may read garbage input from memory that it shouldn't be reading.

Returning to the `Flanger` example created earlier, you may wish to add rate checking to that class. In fact, since the `Flanger` is a kind of filter, you might find it sensible to use the `checkSameRateAsFirstInput` approach, either directly or by modifying the class so it subclasses `Filter` rather than `UGen`.

29.5 Writing the C++ Code

29.5.1 Build Environments

UGen plug-ins are built just like any other C++ project. The recommended way is to use CMake (https://cmake.org/), which is a de facto industry standard. `CMake` itself is not a build system, but rather a build system generator, meaning that it will create the appropriate files depending on whether you prefer to work with Unix makefiles, Xcode projects, Visual Studio (VS) solutions, etc. CMake project files are written in the CMake language and are always named CMakeLists.txt.

First, you need to download/clone the SC source code (https://github.com/supercollider/supercollider/), if you haven't already. For very simple projects, you could just take one of the CMakeLists.txt files from the example plug-ins project (https://github.com/supercollider/example-plugins; e.g., *example-plugins/01a-BoringMixer/CMakeLists.txt*) and in the first line, change the FILENAME variable to the name of your .cpp file.

Alternatively, you can use the excellent cookie-cutter template by Moss Heim (https://github.com/supercollider/cookiecutter-supercollider-plugin). After filling out a short "questionnaire," it will generate the CMakeLists.txt file, along with sample *.cpp, *.sc and *.schelp files. The README provides all the necessary information. Give it a try!

These are the required steps to actually build your project. We will use the command line, but you can also use a graphical frontend such as cmake-gui.

1. Go into the folder containing the CMakeFiles.txt.
2. Create a new directory called "build," which will contain the build results:

```
mkdir ./build
```

3. Change to the new "build" folder:

```
cd ./build
```

4. Now we can run cmake to generate the build system files for us.

We must provide the desired generator with `-G<generator>`. With Linux and MacOS—but also with Windows when compiling with MinGW—you would typically use "Unix Makefiles." For the Visual Studio compiler, you would use "Visual Studio <version> <year> <arch>" instead. You can run `cmake -h` to get a list of all the available generators.

We may also set some options with `-D<option=value>`. For example, you should always specify the build type with `-DCMAKE_BUILD_TYPE=<buildtype>`. For UGen plug-ins, we also need to provide the path to the SC source code with `-DSC_PATH=<path>`. Another common option is to enable Supernova support with `-DSUPERNOVA=ON`.

The last argument to "cmake" is the location of the CMakeFiles.txt file. Typically, it is one level above the build folder, so we can just use a double dot.

Here, we generate Unix makefiles for GCC/clang for a release build with Supernova support:

```
cmake -G"Unix Makefiles" -DCMAKE_BUILD_TYPE=Release   -DSUPERNOVA=O
N ..
```

5. Finally, we can build the project:

```
cmake-build . --config Release
```

6. [Optional] Install the project. (This action is currently not supported in the example-plugins templates.)

```
cmake-install . --config Release
```

During development, you might want to work with a Debug build instead. Not only will it generate debug symbols, it will also compile the source code almost verbatim, without removing any variables or functions, which can make it easier to debug. You just have to replace "Release" with "Debug" in the steps given previously:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake-build . --config Debug
```

29.5.2 When Your Code Will Be Called

The `Server` (scsynth or supernova) will call your plug-in code at four distinct points:

When the `Server` boots, it calls the plug-in's `load()` function, which registers new UGens with the `Server`.

When a UGen is instantiated (i.e., when a `Synth` starts playing), the `Server` calls the UGen's constructor function to perform the setting up of the UGen.

To produce sound, the `Server` calls each UGen's calculation function in turn, *once for every control period*. This is typically the function which does most of the interesting work in the UGen. Since it is called only once during a control period, this function must produce either a single control-rate value or a whole block's worth of audio-rate values. (Note: Demand UGens don't quite fit this description and will be covered later in this chapter.)

When a `Synth` ends, some UGens may need to tidy up, such as freeing memory. If so, these UGens provide a destructor function, which is called at this point.

Additionally, a UGen plug-in might optionally export an `unload()` function which is called before the `Server` shuts down, but this is hardly ever needed. One example is DiskIO_UGens.cpp, where the unload() function stops and joins the disk I/O thread. (Doing this in a global object destructor can deadlock under certain circumstances, especially in Windows DLLs.)

Finally, plug-ins export two more functions without our knowledge: `server_type()` returns 0 for scsynth and 1 for supernova, while `api_version()` returns the plug-in API version number. The `Server` actually calls these before the `load()` function to determine whether the plug-in is compatible and can be loaded. Generally, scsynth will disregard all supernova UGen plug-ins, and vice versa.

29.5.3 The C++ Code for a Basic UGen

The code in figure 29.8 shows the key elements that need to be included in our `Flanger` plug-in code.

```
#include "SC_PlugIn.h"

static InterfaceTable *ft;

// The struct will hold the state of our plugin.
// It is first initialized in the constructor function and then acc
essed
// and possible mutated in every call to the call function.
struct Flanger : public Unit {
   // it is a convention to use some kind of prefix (or postfix)
   // to distinguish member variables from local variables
   float mRate;
   float mDelaysize;
   float mAdvance;
```

```
   float mReadpos;
   int mWritepos;
};

// forward declaration
void Flanger_next(Flanger *unit, int inNumSamples);

void Flanger_Ctor(Flanger *unit) {
  // Here we must initialise *all* state variables in our Flanger s
truct.
  unit->mDelaysize = SAMPLERATE * 0.02f; // Fixed 20ms max delay
  float rate = IN0(1); // initial rate
      // Rather than using rate directly, we're going to calculate
 the size of
      // jumps we must make each time to scan through the delayline
at "rate"
  float delta = (unit->mDelaysize * rate) / SAMPLERATE;
  unit->mAdvance = delta + 1.0f;
  unit->mRate = rate;
  unit->mWritepos = 0;
  unit->mReadpos = 0;

      // IMPORTANT: This tells scsynth the name of the calculation
 function for this UGen.
      SETCALC(Flanger_next);

      // Should also calc 1 sample's worth of output - ensures each
ugen's "pipes" are "primed"
      Flanger_next(unit, 1);

      // store them back
      unit->mRate = rate;
      unit->mAdvance = advance;
      unit->mWritepos = writepos;
      unit->mReadpos = readpos;
}

void Flanger_next(Flanger *unit, int inNumSamples) {
         float *in = IN(0);
         float *out = OUT(0);

      // "rate" and "depth" can be modulated at control rate
      float currate = IN0(1);

      float rate = unit->mRate;
```

```
        float delaysize = unit->mDelaysize;
        float advancce = unit->mAdvance;
        float readpos = unit->mReadpos;

        int writepos = unit->mWritepos;

        for (int i = 0; i < inNumSamples; ++i) {
            float val = in[i];

                    // Do something to the signal before outputting
                    // (not yet done)

                    out[i] = val;
        }
}

PluginLoad(InterfaceTable *inTable) {
ft = inTable;

DefineSimpleUnit(Flanger);
}
```

Figure 29.8
`Flanger` UGen first version, in C code.

Here is what this code does:

First, the `#include` command includes the main header file for SuperCollider's plug-in interface, *SC_PlugIn.h*. This is sufficient to include enough SuperCollider infrastructure for most types of UGen. (For phase vocoder UGens, more may be needed, as described later.) The SuperCollider plug-in interface is mostly a C-style interface (but not a pure C interface). *SC_PlugIn.hpp*, on the other hand, provides the `SCUnit` class, which is essentially a C++ wrapper around the `Unit` struct in *SC_Unit.h*. In the following chapter, we will use the C interface for the sake of simplicity, but also mention the corresponding C++ wrapper functions/methods. The static `InterfaceTable` pointer is a reference to a table of SuperCollider functions.

We define a data structure (a "struct") which will hold any data we need to store during the operation of the UGen. Note that the struct inherits from the base struct `Unit`—this is necessary so the `Server` can correctly write information into the struct, such as pointers to the input and output buffers or the rate at which the unit is running.

In a given plug-in, we are allowed to define one or more UGens. Each of these will have one constructor ("Ctor") function, one or more calculation ("next") functions, and optionally one destructor ("Dtor") function.

Our constructor function, `Flanger_Ctor()`, takes a pointer to a `Flanger` struct and must prepare the UGen for execution. It must do the following 3 things:

1. Initialize the `Flanger` struct's member variables appropriately. In this case, we initialize the `mDelaysize` member to a value representing a 20-ms maximum delay, making use of the `SAMPLERATE` macro which the SuperCollider API provides to specify the sample rate for the UGen. For some of the other struct members, we wish to calculate the values based on an input to the UGen. We can do this using the `IN0()` macro, which grabs a single control-rate value from the specified input. Here, we use `IN0(1)`, remembering that numbering starts at 0; this corresponds to the second input, defined in the `Flanger` class file as "rate." These macros (and others) will be discussed later. Finally, we must set the `mWritepos` and `mReadpos` members to 0 so they have well-defined values. Forgetting to initialize members or variables is a common mistake that can lead to strange behavior or even crashes.
2. Tell the `Server` what the calculation function will be for this instance of the UGen. The `SETCALC` macro stores a reference to the function in our unit's struct. In our example, there's only one choice, so we simply call `SETCALC(Flanger_next)`. It's possible to define multiple calculation functions and allow the constructor to decide which one to use. This is covered later.
3. Calculate one sample's worth of output, typically by calling the unit's calculation function and asking it to process one sample. The purpose of this is to prime the inputs and outputs of all the UGens in the graph and to ensure that the constructors for UGens farther down the chain have their input values available so they can initialize correctly.

With the C++ wrapper, you would use SCUnit's `set_calc_function()` method instead of the `SETCALC` macro. It takes a pointer to a member function and automatically processes one sample.

Our calculation function, `Flanger_next()`, should perform the main audio processing. In this example, it doesn't actually alter the sound—we'll get to that shortly—but it illustrates some important features of calculation functions. It takes two arguments passed in by the server: a pointer to the struct and an integer specifying how many values are to be processed (this will be 1 for the control rate and more—typically 64—for the audio rate).

The last thing in our C++ file is the `load()` function, called when the `Server` executable boots up. For this, we can use the `PluginLoad` macro, which secretly defines and exports the `load()` function as well as the `server_type()` and `api_version()` functions, as explained in the previous section.

In the body of `PluginLoad`, we store the reference to the interface table which is passed in. Note that although you don't see any explicit references to `ft` elsewhere in

the code, that's because they are hidden behind macros which make use of it to call functions in the `Server`.

We must also declare to the `Server` each of the UGens defined by our plug-in. This is done using a macro `DefineSimpleUnit(Flanger)`, which tells the `Server` to register a UGen with the name `Flanger` and with a constructor function named `Flanger_Ctor`. It also tells the server that no destructor function is needed. If we did require a destructor, we would instead use `DefineDtorUnit(Flanger)`, which tells the server that we've also supplied a destructor function named `Flanger_Dtor`. You must name your constructor/destructor functions in this way since the naming convention is hardcoded into the macros. With the C++ wrapper, you only need to call the `registerUnit` function template, which will simply use the constructor and destructor of your C++ class.

So what is happening inside our calculation function? Although in our example the input doesn't actually get altered before being output, the basic pattern for a typical calculation function is given. We do the following:

Create pointers to the input and output arrays which we will access: `float *in = IN(0); float *out = OUT(0);` the macros `IN()` and `OUT()` return appropriate pointers for the desired inputs/outputs—in this case, the first input and the first output. If the input is audio rate, then `in[0]` will refer to the first incoming sample, `in[1]` to the next incoming sample, and so on. If the input is control rate, then there is only one incoming value: `in[0]`.

We use the macro `IN0()` again to grab a single control-rate value, here the "depth" input. Note that IN0() is actually a shortcut to the first value in the location referenced by IN(). `IN0(1)` is exactly the same as *`IN(1)[0]`*.

These macros assume that the `Unit *` parameter is named "unit." With the C++ wrapper, you would rather use the `in()`, `in0()`, and `out()` methods instead.

We copy all the values from the UGen's struct that we need into local variables. Not only does it make the code more readable, it also can improve the efficiency of the unit. We will talk more about this in section 29.7.2.

Next, we loop over the number of input frames, each time taking an input value, processing it, and producing an output value. We could take values from multiple inputs and even produce multiple outputs, but in this example, we're using only one full-rate input and producing a single output.

It is important to note that if an input/output is control rate and you mistakenly treat it as audio rate, you will be reading/writing memory that you should not be, and this can cause bizarre problems and crashes; essentially, this is just the classic C/C++ "gotcha" of accidentally treating an array as being bigger than it really is. Note that in our example, we assume that the input and output are the same size, although it's possible that they aren't; some UGens can take audio-rate input and produce control-rate output.

This is why it is useful to make sure that your SuperCollider class code includes the rate-checking code described earlier in this chapter. You can see why the `checkSameRateAsFirstInput` approach is useful in this case.

The server uses a "buffer coloring" algorithm to minimize the use of buffers and to optimize cache performance. This means that any of the output buffers may be the same as one of the input buffers. This allows in-place operation, which is very efficient. You must be careful, however, not to write any output sample before you have read the corresponding input samples. If you break this rule, then the input may be overwritten with output, leading to undesired behavior. If you can't write the UGen efficiently without breaking this rule, then you can instruct the server not to alias the buffers by using the `DefineSimpleCantAliasUnit()` or `DefineDtorCantAliasUnit()` macros in `PluginLoad` rather than the `DefineSimpleUnit()` or `DefineDtorUnit()` macro. (The Help file on writing UGens provides an example in which this ordering is important.) With the C++ wrapper, you would have to call `registerUnit()` with `disableBufferAliasing` set to `true`.

Finally, having produced our output, we may have modified some of the variables that we loaded from the struct; we need to store them back to the struct so the updated values are used next time. Here, we store the rate value back to the struct. Although we don't modify it in this example, we will shortly change the code so that this may happen.

The code in figure 29.8 should compile correctly into a plug-in. With the class file in place and the plug-in compiled, you can now use the UGen in a `Synth` graph.

```
s.boot
(
x = {
   var son = Saw.ar([100, 150, 200]).mean;
   var out = Flanger.ar(son);
   out.dup * 0.2;
}.play(s);
)
```

Figure 29.9
`Flanger` in action.

Remember that `Flanger` doesn't currently add any effect to the sound. But we can at least check that it runs correctly (outputting its input unmodified and undistorted) before we start to make things interesting.

29.5.4 Summary: The Three Main Rates of Data Output

Our example has taken input in 3 different ways:

1. Using `IN0()` or for C++ `in0()` in the constructor to take an input value and store it to the struct for later use. Since this reads a value only once, the input is being treated as a *scalar-rate* input.

2. Using `IN0()` or for C++ `in0()` in the calculation function to take a single input value. This treats the input as *control-rate*.

3. Using `IN()` or for C++ `in()` in the calculation function to get a pointer to the whole array of inputs. This treats the input as *audio-rate*. Typically, the size of such an input array is accessed using the `inNumSamples` argument, but note that if you create a control-rate UGen with audio-rate input, then `inNumSamples` will be wrong (it will be 1), so you should instead use the macro `FULLBUFLENGTH` (see table 29.1).

Table 29.1
Useful macros for UGen writers, part I

Macro	C++ Method	Description
`IN(index)`	`in(index)`	A `float*` pointer to input number `index`
`OUT(index)`	`out(index)`	A `float*` pointer to output number `index`
`IN0(index)`	`in0(index)`	A single (control-rate) value from input number `index`
`OUT0(index)`	`out0(index)`	A single (control-rate) value at output number `index`
`INRATE(index)`	`inRate(index)`	The rate of input `index`, an integer value corresponding to one of the following constants: • `calc_ScalarRate` (scalar-rate) • `calc_BufRate` (control-rate) • `calc_FullRate` (audio-rate) • `calc_DemandRate` (demand-rate)
`SETCALC(func)`	`set_calc_function`	Set the calculation function to *func*. NB: `set_calc_function()` automatically calculates one sample!
`SAMPLERATE`	`sampleRate()`	The sample rate of the UGen as a double. Note: for control-rate UGens, this is not the full audio-rate, but the audio-rate/block size
`SAMPLEDUR`	`sampleDur()`	Reciprocal of `SAMPLERATE` (seconds per sample)
`BUFLENGTH`	`bufferSize()`	Equal to the block size if the unit is audio-rate and to 1 if the unit is control-rate
`BUFRATE`	`controlRate()`	The control-rate as a double
`BUFDUR`	`controlDur()`	The reciprocal of `BUFRATE`
`FULLRATE`	`fullSampleRate()`	The full audio sample rate of the server (regardless of the rate of the UGen) as a double
`FULLBUFLENGTH`	`fullBufferSize()`	The integer number of samples in an audio-rate input (irrespective of the rate of the UGen)

If a UGen input is actually audio rate, there is no danger in treating it as control rate or scalar rate by reading only the first sample. The result would be crude downsampling without low-pass filtering, which may be undesirable, but it will not crash the server. Similarly, a control-rate input can safely be treated as scalar-rate.

29.5.5 Allocating Memory and Using a Destructor

Next, we can develop our `Flanger` example so it applies an effect to the sound. In order to create a flanging effect, we need a short delay line (around 20 ms). We vary the amount of delay and mix the delayed sound with the input to produce the effect.

To create a delay line, we need to allocate some memory and store a reference to that memory in the UGen's data structure. And, of course, we need to free this memory when the UGen is freed. This requires a UGen with a destructor. Figure 29.10 shows the full code, with the destructor added, as well as the code to allocate, free, and use the memory. Note the change in PluginLoad—we use `DefineDtorUnit()` rather than `DefineSimpleUnit()`. (We've also added code to the calculation function, which reads and writes to the delay line, creating the flanging effect.)

```
#include "SC_PlugIn.h"

static InterfaceTable *ft;

// The struct will hold the state of our plugin.
// It is first initialized in the constructor function and then acc
essed
// and possible mutated in every call to the call function.
struct Flanger : public Unit {
    // it is a convention to use some kind of prefix (or postfix)
    // to distinguish member variables from local variables
    float mRate;
    float mDelaysize;
    float mAdvance;
    float mReadpos;
    int mWritepos;
        // a pointer to the memory we'll use for our internal delay
    float *mDelayline;
};

// forward declaration
void Flanger_next(Flanger *unit, int inNumSamples);

void Flanger_Ctor(Flanger *unit) {
    // Here we must initialise *all* state variables in our Flanger
struct.
    unit->mDelaysize = SAMPLERATE * 0.02f; // Fixed 20ms max delay
    float rate = IN0(1); // initial rate
    // Rather than using rate directly, we're going to calculate th
e size of
```

```
    // jumps we must make each time to scan through the delayline a
t "rate"
    unit->mAdvance = ((unit->mDelaysize * rate) / SAMPLERATE) + 1.0
f;
    unit->mRate = rate;
    unit->mWritepos = 0;
    unit->mReadpos = 0;
        // Allocate the delay line
    unit->mDelayline = (float*)RTAlloc(unit->mWorld, (int)unit->mDe
laysize * sizeof(float));
    // Check the result of RTAlloc because it can fail if the RT po
ol is too
// small!
    if (!unit->mDelayline)
    {
       Print("RTAlloc failed!");
       // clear outputs, set calc function and set done
       ClearUnitOutputs(unit, 1);
       SETCALC(ClearUnitOutputs);
       unit->mDone = true;
       return;
    }
    // Set the delay line to zeros.
    memset(unit->mDelayline, 0, unit->mDelaysize * sizeof(float));

       // IMPORTANT: This tells scsynth the name of the calculation
function
// for this UGen.
       SETCALC(Flanger_next);

       // Should also calc 1 sample's worth of output - ensures eac
h ugen's
// "pipes" are "primed"
       Flanger_next(unit, 1);
}

void Flanger_next(Flanger *unit, int inNumSamples) {
        float *in = IN(0);
    float *out = OUT(0);

    // "rate" and "depth" can be modulated at control rate
    float currate = IN0(1);
        float depth = IN0(2);

    // The compiler doesn't know that "out" can't possibly point
```

```
    // to one of our members, so it would have to reload them from
    // memory on every loop iteration. To prevent this from happeni
ng,
    // we temporarily store them in local variables.
    float rate = unit->mRate;
    float advance = unit->mAdvance;
    float readpos = unit->mReadpos;
    int writepos = unit->mWritepos;
    const float delaysize = unit->mDelaysize; // this one is fixed
    float *delayline = unit->mDelayline;

    if (rate != currate) {
               // rate input needs updating
               rate = currate;
       advance = ((delaysize * rate * 2) / SAMPLERATE) + 1.0f;
       }

for (int i = 0; i < inNumSamples; ++i) {
    float val = in[i];

               // Write to the delay line
               delayline[writepos++] = val;
    if(writepos == delaysize)
                writepos = 0;

               // Read from the delay line
    float delayed = delayline[(int)readpos];
    readpos += advance;
               // Update position, NB we may be moving forwards or
 backwards
// (depending on input)
    while(readpos >= delaysize)
                 readpos -= delaysize;
    while(readpos < 0)
                 readpos += delaysize;

               // Mix dry and wet together, and output them
               out[i] = val + (delayed * depth);
   }

   // store them back
   unit->mRate = rate;
   unit->mAdvance = advance;
   unit->mWritepos = writepos;
   unit->mReadpos = readpos;
```

```
}

void Flanger_Dtor(Flanger *unit) {
   // check in case RTFree failed in the Ctor
   if (unit->mDelayline)
       RTFree(unit->mWorld, unit->mDelayline);
}

PluginLoad(InterfaceTable *inTable) {
   ft = inTable;

   // our Unit has a destructor!
   DefineDtorUnit(Flanger);
}

#include "SC_PlugIn.hpp"

static InterfaceTable *ft;

// The struct will hold the state of our plugin.
// It is first initialized in the constructor function and then acc
essed
// and possible mutated in every call to the call function.
class Flanger : public SCUnit {
public:
    Flanger();
    ~Flanger();
    void next(int inNumSamples);
private:
    // it is a convention to use some kind of prefix (or postfix)
    // to distinguish member variables from local variables
    float mRate;
    float mDelaysize;
    float mAdvance;
    float mReadpos;
    int mWritepos;
    // a pointer to the memory we'll use for our internal delay
    float *mDelayline;
};

Flanger::Flanger() {
    // Here we must initialise *all* state variables in our Flanger
struct.
    mDelaysize = sampleRate() * 0.02f; // Fixed 20ms max delay
    float rate = IN0(1); // initial rate
```

```
    // Rather than using rate directly, we're going to calculate th
e size of
    // jumps we must make each time to scan through the delayline a
t "rate"
    mAdvance = ((mDelaysize * rate) / sampleRate()) + 1.0f;
    mRate = rate;
    mWritepos = 0;
    mReadpos = 0;

    // Allocate the delay line
    mDelayline = (float*)RTAlloc(mWorld, (int)mDelaysize * sizeof(f
loat));
    // Check the result of RTAlloc because it can fail if the RT po
ol is too
// small!
    if (!mDelayline)
    {
        Print("RTAlloc failed!");
        // clear outputs, set calc function and set done
        ClearUnitOutputs(unit, 1);
        SETCALC(ClearUnitOutputs);
        mDone = true;
        return;
    }
    // Set the delay line to zeros.
    memset(mDelayline, 0, mDelaysize * sizeof(float));

    // sets the calc function and automatically computes 1 sample.
    set_calc_function<&Flanger::next>();
}

Flanger::~Flanger() {
    // check in case RTFree failed in the Ctor

    if (mDelayline)
        RTFree(mWorld, mDelayline);
}

void Flanger::next(int inNumSamples ) {
    float *in = in(0);
    float *out = out(0);

    // "rate" and "depth" can be modulated at control rate
    float currate = in0(1);
    float depth = in0(2);
```

```
    // The compiler doesn't know that "out" can't possibly point
    // to one of our members, so it would have to reload them from
    // memory on every loop iteration. To prevent this from happeni
ng,
    // we temporarily store them in local variables.
    float rate = mRate;
    float advance = mAdvance;
    float readpos = mReadpos;
    int writepos = mWritepos;
    const float delaysize = mDelaysize; // this one is fixed
    float *delayline = mDelayline;

    if (rate != currate) {
        // rate input needs updating
        rate = currate;
        advance = ((delaysize * rate * 2) / SAMPLERATE) + 1.0f;
    }

    for (int i = 0; i < inNumSamples; ++i) {
        float val = in[i];

        // Write to the delay line
        delayline[writepos++] = val;
        if(writepos == delaysize)
            writepos = 0;

        // Read from the delay line
        float delayed = delayline[(int)readpos];
        readpos += advance;
        // Update position, NB we may be moving forwards or backwar
ds
// (depending on input)
        while(readpos >= delaysize)
           readpos -= delaysize;
        while(readpos < 0)
           readpos += delaysize;

        // Mix dry and wet together, and output them
        out[i] = val + (delayed * depth);
    }

    // store them back
    mRate = rate;
    mAdvance = advance;
```

```
    mWritepos = writepos;
    mReadpos = readpos;
}

PluginLoad(InterfaceTable *inTable) {
    ft = inTable;

    registerUnit<Flanger>(ft, "Flanger");
}
```

Figure 29.10
`Flanger` UGen second version, C code.

SuperCollider UGens allocate memory differently from most programs. Ordinary memory allocation and freeing can be a relatively expensive operation, so SuperCollider provides a *real-time pool* of memory from which UGens can borrow chunks in an efficient manner. The functions to use in a plug-in are in the right column of table 29.2, and the analogous functions (the ones to avoid) are shown in the left column.

Table 29.2
Memory allocation and freeing

Typical C Memory Management Function	In SuperCollider (using the real-time memory pool)
`void *ptr = malloc(numbytes)`	`void *ptr = RTAlloc(unit->mWorld, size)`
`void *ptr = realloc(ptr, newsize)`	`void *ptr = RTRealloc(unit->mWorld, newsize)`
`free(ptr)`	`RTFree(unit->mWorld, ptr)`

`RTAlloc` and `RTFree` can be called anywhere in your constructor/calculation/destructor functions. Often, you will `RTAlloc` the memory during the constructor and `RTFree` it during the destructor, as shown previously in figure 29.2. Allocating and freeing real-time memory repeatedly in the calculation function almost always constitute a mistake because you can either preallocate a large enough buffer in the constructor or—if you're careful—allocate on the stack with `alloca()`.

Memory allocated in this way is taken from the limited real-time pool and is not accessible outside the UGen (e.g., to client-side processes). If you require large amounts of memory or wish to access the data from the client, you may prefer to use a `Buffer`; this idea is described later.

Warning: Because the size of the real-time memory pool is limited, `RTAlloc` can fail! This means that you always have to check the result of `RTAlloc` for `NULL`. It is recommended to use the `ClearUnitIfMemFailed` macro (see table 29.2). Note that it is

also good practice to check for `NULL` in the destructor before attempting to free the memory with `RTFree`.

29.5.6 Providing More Than One Calculation Function

Your UGen's choice of calculation function is specified within the constructor rather than being fixed. This gives an opportunity to provide different functions optimized for different situations (e.g., one for control-rate and one for audio-rate input) and to decide which one to use. The code in figure 29.11, to be used in the constructor, would choose between two calculation functions according to whether the first input was audio-rate or not.

```
if (INRATE(0) == calc_FullRate) {
   SETCALC(Flanger_next_a);
} else {
   SETCALC(Flanger_next_k);
}
```

Figure 29.11
Choosing a calculation function.

You would then provide both `Flanger_next_a()` and `Flanger_next_k()` functions.

Similarly, you could specify different calculation functions for audio-rate versus control-rate output (e.g., by testing whether `BUFLENGTH` is 1; see table 29.2), although this is often catered to automatically when your calculation function uses the `inNumSamples` argument to control the number of loops performed, and so on.

The unit's calculation function can also be changed during execution—the `SETCALC()` macro can safely be called from a calculation function, not just from the constructor. Whenever you call `SETCALC()`, this changes which function the server will call from the next control period onward.

The Help file on writing UGens shows more examples of `SETCALC()` in use.

29.5.7 Trigger Inputs

Many UGens use trigger inputs. The convention here is that if the input is nonpositive (i.e., 0 or negative) and then crosses to any positive value, a trigger has occurred. If you wish to provide trigger inputs, use this same convention.

The change from nonpositive to positive requires checking the trigger input's value against its previous value. This means that our struct will need a member to store the previous value for checking. Assuming that our struct contains a float member `prevtrig`, figure 29.12 outlines how we handle the incoming data in our calculation function.

```
float trig = IN0(3); // Or whichever input you wish
float prevtrig = unit->prevtrig;
if (prevtrig <= 0 && trig > 0) {
   //. . .do something. . .
}
unit->prevtrig = trig; // Store current value--next time it'll   //
 be the "previous" value
```

Figure 29.12
Control rate trigger input.

The sketch is for a control-rate trigger input, but a similar approach is used for audio-rate triggering too. For audio-rate triggering, you need to compare each value in the input block against the value that came immediately before it. Note that for the very first value in the block, you need to compare against the last value from the *previous* block (which you must have stored).

For complete code examples, look at the source of the `Trig1` UGen, found in *TriggerUGens.cpp* in the main SC distribution.

29.5.8 Accessing a Buffer

When a `Buffer` is allocated and then passed in to a UGen, the UGen receives the index number of that `Buffer` as a float value. In order to get a pointer to the correct chunk of memory (as well as the size of that chunk), the UGen must look it up in the server's list of `Buffers`.

In practice, this is most easily achieved by using the `GET_BUF` and `GET_BUF_SHARED` macros. Use the first if you want to *write* to a Buffer and the second when you only want to *read* from a Buffer. For scsynth, both macros are actually the same, but for Supernova, there is a difference. (Supernova protects shared resources like Buffers or Busses with reader-writer-spinlocks, which means that several readers can access the same resource concurrently, but writers always have exclusive access.) You can call `GET_BUF` or `GET_BUF_SHARED` near the top of your calculation function, and then the data is available via a float pointer `bufData`, along with two integers defining the size of the buffer, `bufChannels` and `bufFrames`. Note that the macro assumes that the buffer index is the *first* input to the UGen (this is the case for most UGens that use a Buffer). If, for some reason, you need to pass the Buffer number to a different UGen input, you can just copy the code in the macro and modify it accordingly.

For examples, look at the code for the `BufRd` or `BufWr` UGens defined in *DelayUGens.cpp* in the SC repository. Note that the former uses `GET_BUF_SHARED`, while the latter uses `GET_BUF`.

Your UGen must not attempt to free the memory associated with a Buffer once it ends. The memory is managed externally by the Buffer allocation/freeing of server commands.

29.5.9 Randomness

The API provides a convenient interface for accessing good-quality pseudorandom numbers. The randomness API is specified in *SC_RGen.h* and provides functions for random numbers from standard types of distribution: uniform, exponential, bilinear, and quasi-Gaussian (such as *sum3rand*, which is also available client-side). The Server creates several independent random number generator instances for UGens to access and stores them in the World structure: `mRGen` points to an array of RGens and `mNumRGens` contains the number of elements. For convenience, the Graph struct has a pointer to the first RGen element. Figure 29.13 shows how to generate random numbers for use in your code.

```
RGen & rgen = *unit->mParent->mRGen;
float rfl = rgen.frand(); // A random float, uniformly   // distrib
uted, 0.0 to 1.0
int rval2 = rgen.irand(56); // A random integer, uniformly   // dis
tributed, 0 to 55 inclusive
float rgaus = rgen.sum3rand(3.5); // Quasi-Gaussian, limited   // t
o range ±3.5
```

Figure 29.13
Random number generator.

29.5.10 When Your UGen Has No More to Do

Many UGens carry on indefinitely, but often a UGen reaches the end of its useful "life" (e.g., it finishes outputting an envelope or playing a buffer). There are three specific behaviors that might be appropriate if your UGen does reach a natural end:

1. Some UGens set a "done" flag to indicate that they've finished. Other UGens can monitor this and act in response to it (e.g., `Done`, `FreeSelfWhenDone`). See the Help files of these UGens for examples. If you wish your UGen to indicate that it has finished, set the flag as follows:

```
unit->mDone = true;
```

This doesn't affect how the server treats the UGen. The calculation function will still be called in the future.

2. In addition, UGens such as `PlayBuf`, `RecordBuf`, `EnvGen`, `Duty`, and `Line` provide a `doneAction` argument which can perform various actions, such as freeing the node once the UGen has reached the end of its functionality. See the Help file for `Done` for a list of all possible doneActions. You can implement this yourself simply by calling the `DoneAction()` macro, which performs the desired action. You would typically allow the user to specify `doneAction` as an input to the unit. For example, if `doneAction` is the sixth input to your UGen, you would call

```
DoneAction(IN0(5), unit)
```

3. Calling `DoneAction()` does not guarantee that your UGen will stop producing sound during future control periods. For example, the user might pass 0 as the `doneAction`—which does nothing! Therefore, you want to make sure that your UGen outputs meaningful values even after it is done. For example, `Line` and `XLine` will simply continue to output their end level after they have finished and called `DoneAction`. If you wish to output zeroes from all outputs of your unit, you can simply set the calculation function to `ClearUnitOutputs`, which provides an irreversible but efficient way for your UGen to produce silent output for the remainder of the `Synth`'s execution.

However, not all UGens remain in the "done" state forever. For instance, `Linen` or `EnvGen` can be (re)triggered with a "gate" input. Others, like `PlayBuf` or `RecordBuf`, allow the user to restart the UGen with a trigger input; they might also provide a "loop" input, which if true, will cause the UGen to restart automatically without setting `mDone` or calling `DoneAction`.

29.5.11 Summary of Useful Macros

Table 29.3 summarizes some of the most generally useful macros defined for use in your UGen code. Many of these are discussed in this chapter, but not all are covered explicitly. The macros are defined in *SC_Unit.h* and *SC_InterfaceTable.h*. If you use the C++ wrapper, many macros in *SC_Unit.h* can be replaced by proper method calls; see `SCUnit` in *SC_PlugIn.hpp*.

Table 29.3
Useful Macros for UGen writers, part II

Macro	Description
`ClearUnitOutputs(unit, inNumSamples)`	A function which sets all the unit's outputs to 0.
`ClearUnitIfMemFailed(condition)`	Handle out-of-memory errors in the UGen constructor. If the given condition is false, the UGen will stop processing and only output silence. Also, `mDone` will be set to `true` (see section 29.5.10). Finally, the macro will cause the current function to return.

Macro	Description
`GET_BUF`	Treats the UGen's first input as a reference to a buffer; looks this buffer up in the server, and provides variables for accessing it, including `float* bufData`, which points to the data; `uint32 bufFrames` for how many frames the buffer contains; `uint32 bufChannels` for the number of channels in the buffer
`GET_BUF_SHARED`	Same as `GET_BUF`, but for read-only access. (This leads to better performance in `Supernova`.)
`Print(fmt,…)`	Print text to the SuperCollider post window; arguments are just like those for the C function `printf`.
`DoneAction(doneAction, unit)`	Perform a `doneAction`, as used in `EnvGen`, `DetectSilence`, and others.
`RTAlloc(world, numBytes)`	Allocate memory from the real-time pool—analogous to `malloc(numBytes)`.
`RTRealloc(world, pointer, numBytes)`	Reallocate memory in the real-time pool—analogous to `realloc(pointer, numBytes)`
`RTFree(world, pointer)`	Free allocated memory back to the real-time pool—analogous to `free(pointer)`.
`SendTrigger(node, triggerID, value)`	Send a trigger from the node to clients, with integer ID `triggerID`, and float value `value`.

29.6 Specialized Types of UGen

29.6.1 Multiple-Output UGens

In the C++ code, writing UGens which produce multiple outputs is very straightforward. The `OUT()` macro gets a pointer to the desired output buffer. Thus, for a three-output UGen, assign each one (`OUT(0)`, `OUT(1)`, `OUT(2)`) to a variable, and then write output to these three pointers. With the C++ wrapper, you would use the `out()` method instead.

In the SuperCollider class code, the default is to assume a single output, and we need to modify this behavior. Let's look at the `Pitch` UGen to see how it's done (see figure 29.14).

```
Pitch: MultiOutUGen {
   *kr {arg in = 0.0, initFreq = 440.0, minFreq = 60.0,   maxFreq =
4000.0, execFreq = 100.0, maxBinsPerOctave = 16, median = 1,
        ampThreshold = 0.01, peakThreshold = 0.5, downSample = 1;
   ^this.multiNew('control', in, initFreq, minFreq, maxFreq,   exec
Freq, maxBinsPerOctave, median, ampThreshold, peakThreshold, downSa
mple)
   }
   init {arg. . .theInputs;
        inputs = theInputs;
      ^this.initOutputs(2, rate);
```

```
    }
}
```

Figure 29.14
`MultiOutUGen`.

There are two differences from an ordinary UGen. First, `Pitch` is a subclass of `MultiOutUGen` rather than of UGen; `MultiOutUGen` takes care of some of the changes needed for a UGen with multiple outputs. Second, the `init` function is overridden to say exactly how many outputs this UGen will provide (in this case, two).

For `Pitch`, the number of outputs is fixed, but in some cases, it might depend on other factors. `PlayBuf` is a good example of this: its number of outputs depends on the number of channels in the buffers that it is expecting to play, specified using the `numChannels` argument. The init method for `PlayBuf` takes the `numChannels` input and specifies that as the number of outputs.

29.6.2 Passing Arrays into UGens

29.6.2.1 The class file As described earlier, the *multiNew* method automatically performs multichannel expansion if any of the inputs are arrays—yet in some cases, we want a single unit to handle a whole array, rather than having one unit per array element. The `BufWr` and `RecordBuf` UGens are good examples of UGens that do exactly this: each UGen can take an array of inputs and write them to a multichannel buffer. Figure 29.15 shows how the class file handles this.

```
RecordBuf: UGen {
    *ar {arg inputArray, bufnum = 0, offset = 0.0, recLevel = 1.0,
    preLevel = 0.0, run = 1.0, loop = 1.0, trigger = 1.0;
        ^this.multiNewList(['audio', bufnum, offset, recLevel,   pr
eLevel, run, loop, trigger] ++ inputArray.asArray);
    }
}
```

Figure 29.15
`RecordBuf`.

Instead of calling the UGen method *multiNew*, we call *multiNewList*, which is the same except that all the arguments are a single array rather than a separated argument list. The *inputArray* argument (which could be either a single unit or an array) is concatenated onto the end of the argument list using the ++ array concatenation operator, and therefore it appears as a set of *separate* input arguments rather than a single array argument.

Note that RecordBuf doesn't know in advance what size the input array is going to be. Because of the array flattening that we perform, the RecordBuf C++ plug-in receives a *variable number of inputs* each time it is instantiated. Our plug-in code will be able to detect how many inputs it receives in a given instance.

Why do we put *inputArray* at the *end* of the argument list? Why not at the beginning, in parallel with how a user invokes the RecordBuf UGen? The reason is that it makes things simpler for the C++ code, which will access the plug-in inputs according to their numerical position in the list. The *recLevel* input, for example, is always the third input, whereas if we inserted *inputArray* into the list before it, its position would depend on the size of *inputArray*.

The Poll UGen uses a very similar procedure, converting a string of text to an array of ASCII characters and appending them to the end of its argument list. However, the Poll class code must perform some other manipulations, so it is perhaps less clear as a code example than RecordBuf. But if you are developing a UGen that needs to pass text data to the plug-in, Poll shows how to do it using this array approach.

Unfortunately, this strategy essentially breaks multichannel expansion. Consider the following code:

```
BufWr.ar([SinOsc.ar([100, 200, 400]), SinOsc.ar([150, 300,   60
0])], [20, 21])
```

Here, we attempt to write two SinOsc Arrays (with three elements each) into two different Busses. We would naively expect this to be the same as

```
[BufWr.ar(SinOsc.ar([100, 200, 400]), 20), BufWr.ar(SinOsc.   ar([1
50, 300, 600]), 21)]
```

But this is not the case! When you evaluate both lines in the SC integrated development environment (IDE), you will see that the former creates an Array of three(!) BufWr UGens. How does this happen? Remember that in the init method, we concatenate the inputArray with the remaining inputs before passing it to multiNewList (see BufWr.ar in *BufIO.sc*). In our case, the resulting list would essentially look like this:

```
['audio', [20, 21], phase, loop, [SinOsc.ar(100), SinOsc.   ar(20
0), SinOsc.ar(400)], [SinOsc.ar(150), SinOsc.ar(300),   SinOsc.ar(6
00)]]
```

After multichannel expansion, you would get three BufWr instances with two input channels each, writing to Busses 20, 21, and 20 respectively. (The Bus number wraps

around.)

In fact, the Help file for `BufWr` acknowledges this issue by saying that "BufWr … does not do multichannel expansion, because input is an array." However, there is an alternative: You can mandate that users pass the `Array` as a `Ref`! `Refs` do not look like `Arrays`, so they are ignored in multichannel expansion. When you override the init method, you can finally dereference it by calling `value` on it and append the resulting `Array` to the other inputs. The user can pass an `Array` of `Array Refs` and multichannel expansion will work as expected.

In fact, this technique even becomes necessary if you want multidimensional `Arrays` as UGen inputs. For example, `Klank`, `Klang`, and `DynKlank` all require their first input (`specificationsArrayRef`) to be a `Ref` to three `Arrays`.

For one-dimensional array inputs, plain `Arrays` are still recommended. You lose multichannel expansion, but it's an idiom that users are already familiar with. After all, everybody uses the `Out` UGen. `Refs`, on the other hand, are a bit obscure, and it is easy to forget about that little backtick.

29.6.2.2 The C++ code Ordinarily, we access input data using the `IN()` or `IN0()` macro, specifying the number of the input that we want to access. Arrays are passed into the UGen as separate numeric input for each array element, so we access these elements in exactly the same way. But we need to know how many items to expect since the array can have a variable size.

The `Unit` struct can tell us how many inputs in total are being provided (the `mNumInputs` member or the `numInputs()` method). Look again at the `RecordBuf` class code given earlier in this chapter. There are seven "ordinary" inputs, plus the array appended to the end. Thus the number of channels in our input array is (`unit->mNumInputs-7`). We use this information to iterate over the correct number of inputs and process each element.

29.6.3 Demand-Rate UGens

29.6.3.1 The class file Writing the class file for a demand-rate UGen is straightforward. Look at the code for units such as `Dseries`, `Dgeom`, or `Dwhite` as examples. They differ from other UGen class files in two ways:

1. The first argument to multiNew (or multiNewList) is `'demand'`.
2. They implement a single class method, `*new`, rather than `*ar/*kr/*ir`. This is because although some UGens may be able to run at multiple rates (e.g., audio rate or control rate), a demand-rate UGen can run at only one rate: the rate at which data are demanded of it.

29.6.3.2 The C++ code The C++ code for a demand-rate UGen works normally, with the constructor specifying the calculation function. However, the calculation

function behaves slightly differently.

First, it is not called regularly (once per control period), but only when demanded, which during a particular control period could be more than once or not at all. This means that you can't make assumptions about regular timing, such as the assumptions made in an oscillator, which increments its phase by a set amount each time that it is called.

Second, rather than being invoked directly by the Server, the calculation function calls are actually passed up the chain of demand-rate generators. Instead of the `IN()` or `IN0()` macros to access an input value, we use the `DEMANDINPUT()` macro, which requests a new value directly from the unit farther up the chain, on demand.

Note: because of the method used to demand the data, demand-rate UGens are currently restricted to being single output.

29.6.4 Phase Vocoder UGens

Phase vocoder UGens operate on frequency-domain data stored in a buffer (produced by the FFT UGen). They don't operate at a special rate of their own: in reality, they are control-rate UGens. They produce and consume a control-rate signal, which acts as a type of trigger: when an FFT frame is ready for processing, its value is the appropriate buffer index; otherwise, its value is –1. This signal is often referred to as the "chain" in SC documentation.

29.6.4.1 The class file As with demand-rate UGens, phase vocoder UGens (PV UGens) can have only a single rate of operation: the rate at which FFT frames are arriving. Therefore, PV UGens implement only a single `*new` class method, and they specify their rate as "control" in the call to *multiNew*. See the class files for `PV_MagMul` and `PV_BrickWall` as examples

PV UGens process data stored in buffers, and the plug-in API provides some useful macros to help with this. The macros assume that the *first* input to the UGen is the one carrying the FFT chain where data will be read and then written, so it is sensible to stick with this convention.

29.6.4.2 The C++ code PV UGens are structured just like any other UGen, except that to access the frequency-domain data held in the external buffer, certain macros and procedures must be used. Any of the core UGens implemented in *PV_UGens.cpp* should serve as a good example on which base your own UGens. Your code should include the header file *FFT_UGens.h*, which defines some PV-specific structs and macros.

Two important macros are `PV_GET_BUF` and `PV_GET_BUF2`, one of which you use at the beginning of your calculation function to obtain the FFT data from the buffers. These macros implement the special PV UGen behavior: if the FFT chain has "fired," then they access the buffers and continue with the rest of the calculation function; but if the

FFT chain has not fired in the current control block, then they output a value of–1 and *return* (i.e., they do not allow the rest of the calculation function to proceed). This has the important consequence that although your calculation function code will look as if it is called once per control block, in fact your code will be executed only at the FFT frame rate.

`PV_GET_BUF` will take the FFT chain indicated by the *first* input to the UGen and create a pointer to the data called `*buf`. `PV_GET_BUF2` is for use in UGens which process two FFT chains and write the result back out to the first chain: it takes the FFT chain indicated by the *first* and *second* inputs to the UGen and creates pointers to the data called `*buf1` and `*buf2`. It should be clear that you will use `PV_GET_BUF` or `PV_GET_BUF2`, but not both.

Having acquired a pointer to the data, you will wish to read/write that data, of course. Before doing so, you must decide whether to process the complex-valued data as polar coordinates or Cartesian coordinates. The data in the buffer may be in either format (depending on what has happened to them so far). To access the data as Cartesian values, you use

```
SCComplexBuf *p = ToComplexApx(buf);
```

and to access the data as polar values, you use

```
SCPolarBuf *p = ToPolarApx(buf);
```

If the data already have the desired format, these functions just return a pointer to the data; otherwise, they transform the data *in place*. These data structures and functions are declared in *FFT_UGens.h*.

`FFT` data consist of a complex value for each frequency bin, with the number of bins related to the number of samples in the input. But in the SuperCollider context, the input is real-valued data, which means that (1) the bins above the Nyquist frequency (which is half the sampling frequency) are a mirror image of the bins below it, and therefore can be neglected; and (2) phase is irrelevant for the DC and Nyquist frequency bins, so these two bins can be represented by a single-magnitude value rather than a complex value.

The end result of this is that we obtain a data structure containing a single DC value, a single Nyquist value, and a series of complex values for all the bins inbetween. The number of bins in between is given by the value `numbins`, which is provided by `PV_GET_BUF` or `PV_GET_BUF2`. The data in a Cartesian-type struct (an `SCComplexBuf`) take the following form:

```
p->dc
p->bin[0].real
p->bin[0].imag
p->bin[1].real
p->bin[1].imag
. . .
p->bin[numbins-1].real
p->bin[numbins-1].imag
p->nyq
```

The data in a polar-type struct (an `SCPolarBuf`) take the following form:

```
p->dc
p->bin[0].mag
p->bin[0].phase
p->bin[1].mag
p->bin[1].phase
. . .
p->bin[numbins-1].mag
p->bin[numbins-1].phase
p->nyq
```

Note that the indexing is slightly strange: engineers commonly refer to the DC component as the "first" bin in the frequency-domain data. However in these structs, because the DC component is represented differently, bin[0] is actually the first non-DC bin, which would sometimes be referred to as the "second bin." Similarly, keep in mind that `numbins` represents the number of bins *not including* the DC and Nyquist bins.

To perform a phase vocoder manipulation, simply read and write to the struct (which actually directly points into the external Buffer). The buffer will then be passed down the chain to the next phase vocoder UGen. You don't need to do anything extra to output the frequency-domain data.

29.7 Practicalities

29.7.1 Debugging

Standard C++ debugging procedures can be used when developing UGens. The simplest method is to add a line to your code which prints out values of variables—you can use the standard C++ `printf()` method, which in a UGen will print text to the post window.

For more power, you can launch the server process and attach a debugger in order to perform tasks such as pausing the process and inspecting the values of variables. There is a bit of a learning curve, but it is a very handy skill to have! On Linux and MacOS, you would use *gdb* (the GNU debugger) or *lldb* (the LLVM debugger). IDEs like XCode or QtCreator typically offer a nice graphical front end. On Windows, the debugger must match the compiler that you have used to build your UGen (i.e., *gdb* for MinGW or VS's debugger). Note that the Server usually runs as a seperate process.You can boot the Server from the SC IDE as usual and then attach to the Server process from your C++ IDE (or from the terminal). Alternatively, you can launch the Server application directly in your debugger and in the SC IDE connect to it as a remote Serverin the SC IDE. In the latter case, you need to ensure that you start the Server with the correct argument (e.g., -u 57110).

When a UGen crashes, the debugger can show you a "stack trace" or "backtrace" (i.e., a list of functions that were called until the crash occurred). You can then "walk" up the call stack and check local variables to find out where things went wrong. In many C++ IDEs, you can simply hover over the variables in your source code, and you will see a pop-up with the current values.

The most common cause of bugs and crashes is out-of-bound memory access (i.e., writing to memory outside the expected limits). In ordinary C++ programs, accessing memory that has not been allocated yet will crash with a so-called segmentation fault. You can often find the cause of the crash just by looking at the stack trace and local variables. SuperCollider UGens, on the other hand, are allocated from the real-time memory pool. As a consequence, out-of-bound memory access will likely involve memory that has already been allocated. In this case, you won't get a segmentation fault; instead, you might read garbage values—or even worse, overwrite the memory of other UGens! If you find that other UGens start to act in weird ways, this can be a likely cause. Be extra careful when allocating memory with `RTAlloc`: make sure that you allocate the correct size—for instance, it's easy to forget about the `sizeof()` operator—and check that you don't access it outside its limits. You can use the `assert()` macro to test for certain conditions that should always be true:

```
assert(index >= 0 && index < maxSize); // bound checking with   //
 assert()
buffer[index] = value;
```

If the condition is not met, the program terminates with an error message showing the offending line number in the source code. Don't worry about performance—`assert()` statements are ignored for release builds!

In fact, `assert()` relies on the macro `NDEBUG` (= not debugging), which is only defined for release builds. This means that you can write your own handcrafted assertions simply by wrapping the code in an `#ifndef NDEBUG…#endif` block. (Note the double negation!)

Stack buffer overflows are an especially nasty category of bug. They are essentially caused by out-of-bound access to stack memory. On most platforms, the stack grows *downward*. This means that if you write past the *end* of a stack-allocated array, you might overwrite existing call frames. If you accidentally overwrite the return address, the debugger can't produce a meaningful call stack anymore, making these kinds of errors very hard to debug. If the stack appears to be completely wiped out, you might have attempted to clear a variable on the stack with `memset()` but passed a wrong size.

29.7.2 Optimization

Optimizing code is a vast topic that often depends on the specifics of the code in question. However, we can suggest some optimization tips for writing SuperCollider UGens. The efficiency/speed of execution is usually the number-one priority, especially since a user may wish to employ many instances of the UGen simultaneously. The difference between a UGen that takes 2.5 percent of the central processing unit (CPU) and another that takes 1.5 percent may seem small, but the first limits you to 40 simultaneous instances, while the second allows up to 66—a 65 percent increase. Imagine doing your next live performance on a cheap laptop—that's essentially the effect of the less efficient code.

a. Avoid calls to expensive procedures whenever possible. For example, *floating-point division* is typically much more expensive than multiplication, so if your unit must divide values by some constant value which is stored in your struct, rewrite it so that the *reciprocal* of that value is stored in the struct and you can perform a multiplication rather than a division. If you want to find an integer power of 2, use bit shifting ($1 << n$) rather than the expensive math function (pow(2, n)). Other expensive floating-point operations are square root finding and trigonometric operations (e.g., sin, cos, tan). *Precalculate* and store such values in a lookup table wherever possible, rather than calculating them afresh every time the calculation function is called. In fact, *SC_Complex.h* internally uses look-up tables to convert between Cartesian and polar coordinates in an efficient way.

As a typical example, often a filter UGen will take a user parameter (such as cutoff frequency) and use it to derive internal filter coefficients. If you store the previous value of the user parameter and use it to check whether it has changed at all—updating the coefficients only after a change occurs—you can improve efficiency since UGens are often used with fixed or rarely changing parameters.

One of the most important SuperCollider-specific choices is, for reading a certain input or even performing a given calculation, whether to do this at *scalar*, *control*, or *audio rate*. It can be helpful to allow any and all values to be updated at audio rate, but if you find that a certain update procedure is expensive and won't usually be required to run at audio rate, it may be preferable to update only once during a calculation function.

b. Creating *multiple calculation functions*, each appropriate to a certain context (e.g., to a certain combination of input rates, as demonstrated earlier), and choosing the most appropriate ones can allow a lot of optimization. For example, a purely control-rate calculation can avoid the looping required for audio-rate calculation and typically produces a much simpler calculation as a result. There is a maintenance overhead in providing these alternatives, but the efficiency gains can be large. In this tension between efficiency and code comprehensibility/reusability, you should remember the importance of adding comments to your code to clarify the flow and the design decisions that you have made.

c. In your calculation function, store values from your struct as well as input/output pointers/values as *local variables*, especially if referring to them multiple times. We already mentioned this briefly in section 29.5.3, but now is the time to give an explanation: the UGen inputs and outputs are `float*` pointers. *We* know that those will never point to `float` members in our unit, but the compiler does not know this. Therefore, it must assume that writing into an output array might change one of our struct members, so it must reload them from memory on every loop iteration! By temporarily copying all members that we need to local variables, we tell the compiler that they can be changed only by us, so they will be kept in registers whenever possible.

Accessing a member variable `mFoo` in a C++ method is practically the same as doing *unit->mFoo* in a free function. (C++ methods have a hidden `this` pointer.) To be clear, the indirection per se is not the problem. Modern compilers are smart enough to keep variables in registers if they can prove that those can't change unnoticed. (If you're curious, search for "strict aliasing rule" and "C restrict keyword.")

d. Avoid `DefineSimpleCantAliasUnit` *and* `DefineDtorCantAliasUnit` if possible. As described earlier, `DefineSimpleCantAliasUnit` is available as an alternative to `DefineSimpleUnit` in cases where your UGen must write output before it has read from the inputs, but this can decrease cache performance.

e. *Avoid peaky CPU usage.* A calculation function that does nothing for the first 99 times it's called, and then performs a mass of calculations on the 100th call, could cause *audio dropouts* if this spike is very large. To avoid this, "amortize" your unit's effort by spreading the calculation out, if possible, by precalculating some values which are going to be used in that big 100th call.

f. Be aware of *subnormal floating-point numbers*. These are extremely small numbers —and they are *slooow*. Typically, they pop up in recursive algorithms, like IIR filters, when a value converges toward zero. The host application can set certain CPU flags to flush them to zero automatically. To be sure, you might still check and flush values manually at the end of a control block if they come from a feedback path. For this purpose, *SC_InlineUnaryOp.h* contains a very useful function called `zapgremlins()`, which is used extensively in *DelayUGens.cpp* and *FilterUGens.cpp*. If you notice that your plug-in runs about 10x slower (or more) under specific circumstances, check for subnormals in your signal path.

g. *Vectorization.* Modern processors can perform a calculation on multiple pieces of data in a single instruction. The technical term is "SIMD" (which stands for "single instruction, multiple data"). For our use case, we focus on floating-point calculations, but this also works for integers. SSE instructions use 128-bit-wide registers, which means that they can process up to 4 *floats* (4 * 4 * 8 = 128) or 2 *doubles* (2 * 8 * 8 = 128) simultaneously. The newer AVX and AVX2 instructions—introduced in 2011 resp. 2013—extend the register size to 256 bits. Now you can process up to 8 floats simultaneously! Isn't this exciting?

How do you use those fancy instructions? Compilers provide something called "intrinsics"—special statements that refer to low-level CPU operations—but they are different for every instruction set, which makes the code nonportable. Luckily, various people have written SIMD libraries for C++ which hide the difference behind a common interface. SuperCollider uses the *nova-tt* library by Tim Blechmann for many of its core UGens. Documentation is rather sparse, but you can study the source code of some plug-ins. (Just grep for "#define NOVA_SIMD" in the SuperCollider respository.) Since *nova-tt* only consists of header files, you can easily integrate it into your own projects.

Sounds like a lot of work? The good news is, modern compilers are capable of applying these instructions automatically ("auto-vectorization") if you feed them the appropriate compiler flags. (These will be explained in the final section of this chapter.) The compiler will typically add some prologue code to check if the pointers overlap, and if not, jump to a vectorized version of the actual loop.

```
#include <SC_Unit.h>

struct Test : Unit {
    float mMul;
    float mAdd;
};

void Test_next1(Test *unit, int inNumSamples) {
```

```
    float *in = IN(0);
    float *out = OUT(0);
    for (int i = 0; i < inNumSamples; ++i) {
        // mMul and mAdd will be reloaded from memory
        // on every iteration, because the compiler does not
        // know that 'out' can't possible point to them.
        out[i] = in[i] * unit->mMul + unit->mAdd;
    }
}

void Test_next2(Test *unit, int inNumSamples) {
    float *in = IN(0);
    float *out = OUT(0);
    // this tells the compiler that it should keep the variables in
// registers.
    const float mul = unit->mMul;
    const float add = unit->mAdd;
    for (int i = 0; i < inNumSamples; ++i) {
        out[i] = in[i] * mul + add;
    }
}
```

Figure 29.16
Autovectorization.

Figure 29.16 shows a very simple multiply-add UGen with constant `mul` and `add` arguments.

I have compiled this code with GCC, Clang and MSVC using the following flags:

```
GCC and Clang: -O3 -mavx2
MSVC: /O2 /arch:AVX2
```

With these flags, they could all vectorize the code using AVX2 instructions, which means that it would calculate eight multiply-add operations simultaneously!

So why would someone write manual SIMD code in the first place? There are still times when the compiler just isn't clever enough and needs some hand-holding. Also, it makes much of the guess work unnecessary, so it can generate less code. This can lead to better performance, as we will explain now.

h. Use the right *compiler flags*. Compilers produce vastly different code depending on the flags that you pass to them. This already came up when we talked about release versus debug builds in section 29.5.1. All compilers have something called "optimization levels" (i.e., flags which control how aggressively they are allowed to

apply optimizations). At the lowest optimization level (`-O0` for GCC and Clang, `/Od` for MSVC), the source code is compiled almost literally, leading to rather slow code, but it is very useful for debugging. At increasing optimization levels, the compiler will try to reduce expressions, inline function calls, or unroll/vectorize loops (as we have already seen). In fact, many C++ abstractions would not be practical without modern optimizing compilers. The highest optimization levels are `-O2` and `-O3` (GCC/Clang) resp. `/O1` and `/O2` (MSVC). Although one of these flags is set automatically when you configure `CMake` with *-DCMAKE_BUILD_TYPE=Release,* it is better to specify them explicitly.

Although this may seem counterintuitive, sometimes you get faster results by *not* using the highest optimization level. With `-O3` (resp. `/O2`), the compiler will often produce different versions of your code and select the optimal version at runtime. Since it does not have enough knowledge about our program, it will produce things that we won't ever need. The resulting "code bloat" can actually slow the program due to instruction cache misses. In fact, most compilers also offer an optimization level that optimizes for code *size* (e.g., `-Os` (GCC/Clang) or `/Os` (MSVC)). With `CMake`, you can achieve this by setting the build type to *MinSizeRel* instead of *Release*. There are indeed situations where `-Os` or `-O2` leads to overall faster code than `-O3`! The most important rule for writing fast code is: *always measure*!

Another big opportunity for generating faster code are floating-point operations. Most compilers—with the exception of the Intel C++ compiler—adhere to standard conforming IEEE 754 floating-point math by default. Floating-point math has the curious property that it is not associative, meaning that `(a + b) + c` is not necessarily the same as `a + (b + c)`, and `(a * b) * c` is not the same as `a * (b * c)`. The compiler is not allowed to reorder or contract floating-point expressions unless explicitly told to do so. This might be important for scientific computing, but as plug-in authors, we don't care about this level of precision. We want our code to be fast! Luckily, there are several flags to enable so-called unsafe math optimizations. Most important, they lead the compiler to think that floating-point math is associative and no proper rounding is required, so it can group floating-point expressions in the most efficient way. If you want to squeeze out every inch of performance, use `-ffast-math` with GCC and Clang (resp. `/fp:fast` with MSVC). Be warned: with these flags, the compiler pretends that signed zeros, `Inf` (= infinite), and `NaN` (= not a number) don't exist. If your code relies on any of these things, you must use less aggressive flags. For example, `scsynth` and `Supernova` internally use nonsignaling `NaNs` as sentinel values for uninitialized controls or `Buffer` numbers, so they can't be compiled with `-ffast-math` without breaking the code. Most plug-in code, however, is just number crunching, in which case `-ffast-math` makes perfect sense.

Finally, you might want to *tune the code to certain types of CPUs*. By default, compilers generate code that runs on all CPUs with a given instruction set. However,

this means it cannot use any of the newer instructions available. If the CPU detects an instruction that it does not understand, the program will typically terminate with `SIGILL` (= illegal instruction signal). For example, the x86_64 instruction set only mandates SSE2, but most modern CPUs actually have AVX or AVX2 support—which, as we have seen, can be significantly faster! If you do not care about people running your plug-in on 10-year-old CPUs, you can add `-mavx` or `-mavx2` (GCC/Clang) (`resp. /arch:AVX` or `/arch:AVX2` (MSVC)) to your compiler flags. You can also compile for a certain minimum required CPU architecture. For instance, `-march=haswell` stands for the Intel Haswell architecture (2013). For your personal use, you might even set `-march=native` (GCC/Clang), which tunes the code to your very own CPU model. *CMakeLists.txt* files for SuperCollider UGens typically have an option called `NATIVE`, which does exactly this. Be aware that the executable won't be portable—running it on another computer might crash with SIGILL!

29.7.3 Distributing Your UGens

Sharing UGens with others contributes to the SuperCollider community and is a very cool thing to do. In the past, plug-in authors used to add their UGens to the sc3-plugins collection (github.com/supercollider/sc3-plugins/). The repository has grown to a size that essentially became unmaintainable, so it is now in a quasi-frozen state. As of 2024, the last new UGens were added six years ago.

Most people just put their plug-ins on GitHub (or alternatives like GitLab or BitBucket), which means you should probably learn how to use Git as well. Git is a version control system originally developed by Linus Torvalds. There is no space in this book to explain the basics of Git, but there are countless resources online. There is even a complete book available for free at git-scm.com/book/. Git might appear daunting at first, but as a solo developer, you can get away with a few basic operations. All you probably want to do is track changes in your source code, make it accessible from different computers and publish it to some central location like GitHub. You can learn all of this in one or two days—or better, ask one of your programmer friends to show it to you. You will not regret it!

Of course, you can just ask your users to build the plug-in from source, but not everyone has the technical capabilities—or the willingness—to do this. If you want to reach a larger user base, it is a good idea to provide prebuilt binaries for all common platforms. You can use one of the many continuous integration (CI) services, like AppVeyor, Travis, or GitHub Actions. If you are a student, you might be able to use the CI infrastructure of your university. You can then provide the compiled binaries as assets on the release page of your repository.

Unfortunately, SuperCollider does not have a proper package management system for binaries. The Quarks system is really meant for SC class files and accompanying Help

material, but not for plug-in binaries (although a few people do use it for this purpose). Hopefully, in the future, the Quarks system will be extended to deal properly with plug-in binaries.

Remember that SuperCollider is licensed under the well-known GNU Public License, Version 3 (GPLv3) open-source license, including the plug-in API. So if you wish to distribute your plug-ins to the world, they must also be GPLv3-licensed. (Note: you retain copyright in any code you have written. You do not have to sign away your copyright in order to GPL-license a piece of code.) Practically, this has a couple of implications:

- You should include a copyright notice, a copy of the GPLv3 license text, and the source code with your distributed plug-in.
- If your plug-in uses third-party libraries, those libraries must be available under a GPLv3-compatible copyright license. Generally, anything published under a permissive license, such as the MIT or BSD license, falls into this category. See the GNU GPL website for further discussion of what this means.

29.8 Conclusion

This chapter doesn't offer an exhaustive list of all that's possible, but it does provide you with the core of what all UGen programmers need to know. If you want to delve deeper into this topic, you will find the online community to be a valuable resource for answers to questions not covered here, and the source code for existing UGens provides a wealth of useful code examples.

The open-source nature of SuperCollider creates a vibrant online developer community. Whether you are tweaking one of SuperCollider's core UGens or developing something very specialized, you'll find the exchange of ideas with SuperCollider developers can be rewarding for your own projects, as well as for others, and can feed into the ongoing development of SuperCollider as a uniquely powerful and flexible synthesis system.

30 Inside scsynth

Ross Bencina

This chapter explores the implementation internals of scsynth, the server process of SuperCollider 3, which is written in C++. It is intended to be useful to people who are interested in modifying or maintaining the scsynth source code, as well as to those who are interested in learning about the structure and implementation details of one of the great milestones in computer music software. By the time you've finished this chapter, you should have improved your understanding of how scsynth does what it does and also gained some insight into why it is written the way it is. In this chapter, sometimes we'll simply refer to scsynth as "the server." "The client" usually refers to sclang or any other program sending OSC commands to the server. Although the text focuses on the server's real-time operating mode, the information presented here is equally relevant to understanding scsynth's non-real-time mode. As always, the source code is the definitive reference and provides many interesting details, which unfortunately space limitations don't allow us to include here.

Wherever possible, the data, structure, and function names used in this chapter match those in the scsynth source code. Sometimes you may find that the source file, the class name, or both may have an SC_ prefix. I have omitted such prefixes from class and function names in this discussion for consistency.

Also, note that I have chosen to emphasize an object-oriented interpretation of scsynth using UML diagrams to illuminate the code structure, as I believe that scsynth is fundamentally object oriented, if not in an idiomatically C++ way. In many cases, structs from the source code appear as classes in the diagrams. Where appropriate, I have taken the liberty of interpreting inheritance where a base struct is included as the first member of a derived struct. However, I have resisted the urge to translate any other constructs (such as the pseudomember functions mentioned below). All other references to names appear here as they do in the source code.

Now that formalities are completed, in the next section, we set out on our journey through the scsynth implementation with a discussion of scsynth's coding style. Following that, we consider the structure of the code, which implements what I call the scsynth domain model: Nodes, Groups, Graphs, GraphDefs, and their supporting infrastructure. We then go on to consider how the domain model implementation

communicates with the outside world; we consider threading, interthread communications using queues, and how scsynth fulfills real-time performance constraints while executing all the dynamic behaviors offered by the domain model. The final section briefly highlights some of the fine-grained details which make scsynth one of the most efficient software synthesizers on the planet. It is a fantastically elegant and interesting piece of software; I hope you get as much out of reading this chapter as I did in writing it!

30.1 Some Notes on scsynth Coding Style

Although scsynth is coded in C++, for the most part it uses a "C++ as a better C" coding style. Most data structures are declared as plain old C structs, especially those which are accessible to unit plug-ins. Functions which in idiomatic C++ might be considered member functions are typically global functions in scsynth. These are declared with names of the form `StructType_MemberFunctionName (StructType *s[,…])`, where the first parameter is a pointer to the struct being operated on (the "this" pointer in a C++ class). Memory allocation is performed with custom allocators or with `malloc()`, `free()`, and friends. Function pointers are often used instead of virtual functions. A number of cases of what can be considered inheritance are implemented by placing an instance of the base class (or struct) as the first member of the derived struct. There is very little explicit encapsulation of data using getter/setter methods.

There are a number of pragmatic reasons to adopt this style of coding. Probably the most significant is the lack of a completely reliable application binary interface for C++, which makes dynamically linking with plug-ins using C++ interfaces that are compiler version–specific. The avoidance of C++ constructs also has the benefit of making all code operations visible, which in turn makes it easier to understand and predict the performance and real-time behavior of the code.

The separation of data from operations and the explicit representation of operations as data-using function pointers promote a style of programming in which types are composed by parameterizing structs by function pointers and auxilliary data. The use of structs instead of C++ classes makes it less complicated to place objects into raw memory. Reusing a small number of data structures for many purposes eases the burden on memory allocation by ensuring that dynamic objects belong to only a small number of size classes. Finally, being able to switch function pointers at runtime is a very powerful idiom which enables numerous optimizations, as will be seen later in this discussion.

30.2 The scsynth Domain Model

At the heart of scsynth is a powerful yet simple domain model which manages dynamic allocation and evaluation of unit generator graphs in real time. Graphs can be grouped into arbitrary trees whose execution and evaluation order can be dynamically modified (McCartney, 2000). In this section, we explain the main behaviors and relationships between entities in the domain model. The model is presented without concern for how client communication is managed or how the system is executed within real-time constraints. These concerns are addressed in later sections.

Figure 30.1 shows an implementation-level view of the significant domain entities in scsynth. Each class shown on the diagram is a C++ class or struct in the scsynth source code. SC users will recognize the concepts modeled by many of these classes. Interested readers are advised to consult the “ServerArchitecture” section of the Help files for further information about the roles of these classes and the exact operations which can be performed by them.

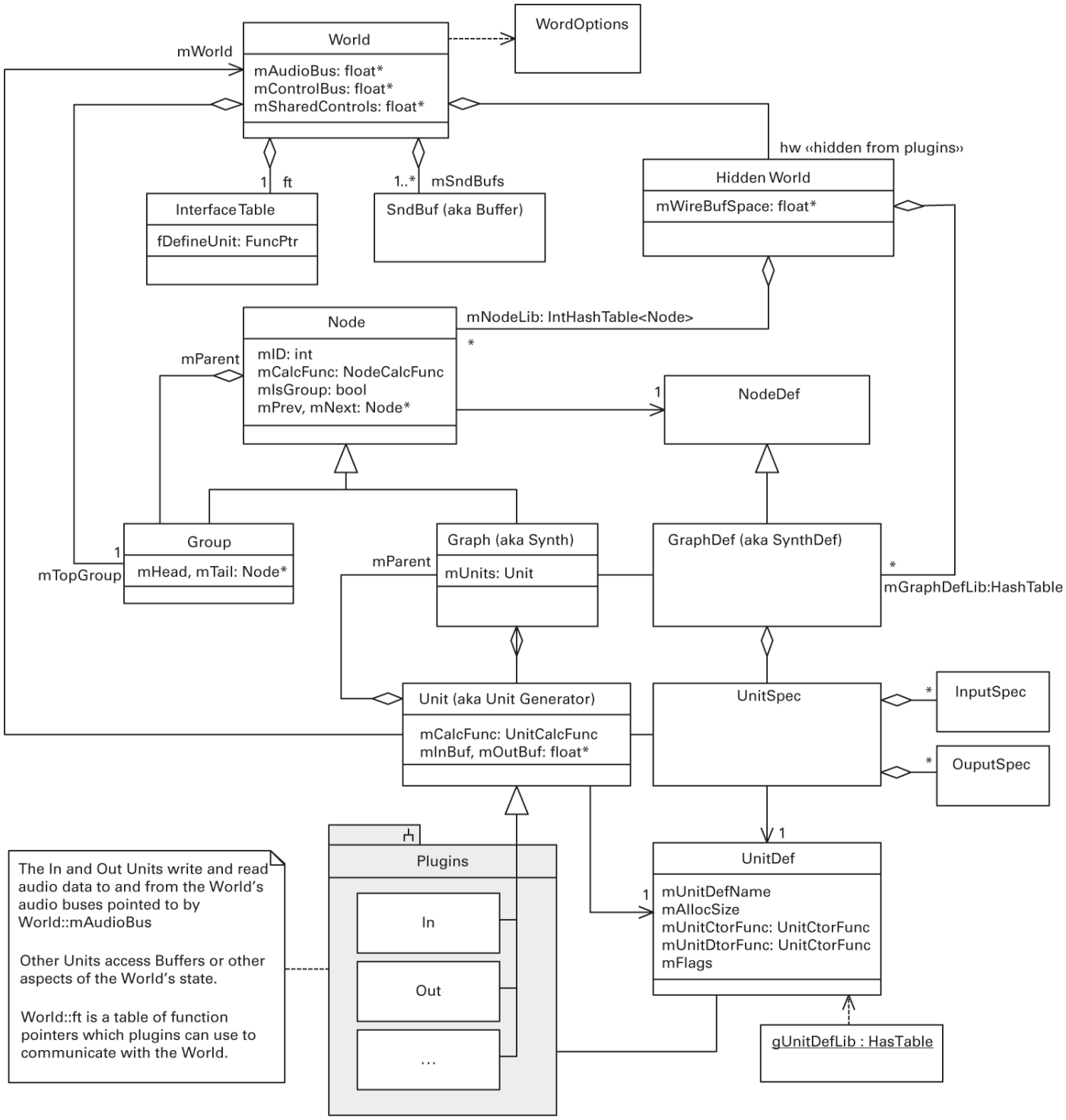


Figure 30.1
Class diagram of significant domain entities.

`World` is the top-level class which (with the exception of a few global objects) aggregates and manages the runtime data in the server. It is created by `World_New()` when scsynth starts. An instance of `WorldOptions` is passed to `World_New()`. It stores the configuration parameters, which are usually passed to scsynth on the command line.

The main task of scsynth is to synthesize and process sound. It does this by evaluating a tree of dynamically allocated `Node` instances (near the middle left of figure 30.1), each of which provides its own `NodeCalcFunc` function pointer, which is called by the server to evaluate the Node at the current time step. `Node::mID` is an integer used by clients to identify specific Nodes in server commands (such as suspending or terminating the Node or changing its location in the tree).

There are two subtypes of `Node`: `Graph` and `Group`. `Graph` is so named because it executes an optimized graph of UGens. It can be likened to a voice in a synthesizer or an "instrument" in a *Music N*-type audio synthesis language such as Csound. The `Graph` type implements the SuperCollider concept of a *Synth*. `Group` is simply a container for a linked list of `Node` instances, and since `Group` is itself a type of `Node`, arbitrary trees may be constructed containing any combination of `Group` and `Graph` instances; readers may recognize this as the *Composite* design pattern (Gamma et al., 1995). The standard `NodeCalcFunc` for a `Group` (`Group_Calc()` in SC_Group.cpp) simply iterates through the `Group`'s contained `Nodes`, calling each `Node`'s `NodeCalcFunc` in turn. Although most code deals with `Nodes` polymorphically, the `Node::mIsGroup` field supports discriminating between `Nodes` of type `Graph` and of `Group` at runtime. Any node can be temporarily disabled using the `/n_run server` command, which switches `NodeCalcFuncs`. When a `Node` is switched off, a `NodeCalcFunc` which does nothing is substituted for the usual one. Disabling a `Group` disables the whole tree under that `Group`.

A `Graph` is an aggregate of interconnected `Unit` subclasses (also known as Unit Generators or UGens). Unit instances are responsible for performing primitive audio DSP operations such as mixing, filtering, and oscillator signal generation. Each `Graph` instance is carved out of a single memory block to minimize the number of expensive calls to the memory allocator. Units are efficiently allocated from the `Graph`'s memory block and evaluated by iterating through a linear array containing pointers to all the `Graph`'s `Units`. Each `Unit` instance provides a `UnitCalcFunc` function pointer to compute samples, which affords the same kind of flexibility as `NodeCalcFunc`. For example, many `Units` implement a form of self-modifying code by switching their `UnitCalcFuncs` on the fly to execute different code paths, depending on their state.

`Graphs` are instantiated using a `GraphDef`, which defines the structure of a class of `Graphs`. The `GraphDef` type implements the SuperCollider concept of *SynthDef*. A `GraphDef` includes both data for passive representation (used on disk and as communicated from clients such as sclang) and optimized in-memory information used to instantiate and evaluate `Graphs` efficiently. `GraphDef` instances store data such as memory allocation size for `Graph` instances, `Unit` initialization parameters, and information about the connections between `Units`. When a new `GraphDef` is loaded into the server, most of the work is done in `GraphDef_Read()`, which converts the

stored representation to the runtime representation. Aside from allocating and initializing memory and wiring in pointers, one of the main tasks that `GraphDef_Read()` performs is to determine which inter-`Unit` memory buffers will be used to pass data between `Units` during Graph evaluation.

The stored `GraphDef` representation specifies an interconnected graph of named `Unit` instances with generalized information about input and output routing. This information is loaded into an in-memory array of `UnitSpec` instances where each `Unit` name is resolved to a pointer to a `UnitDef` (see below), and the `Unit` interconnection graph is represented by instances of `InputSpec` and `OutputSpec`. This interconnection graph is traversed by a graph-coloring algorithm to compute an allocation of inter-`Unit` memory buffers, ensuring that the minimum number of these buffers is used when evaluating the `Graph`. Note that the order of `Unit` evaluation defined by a `GraphDef` is not modified by scsynth.

The tree of `Nodes` in scsynth is rooted at a `Group` referenced by `World::mTopGroup`. `World` is responsible for managing the instantiation, manipulation, and evaluation of the tree of `Nodes`. `World` also manages much of the server's global state, including the buses used to hold control and audio input and output signals (e.g., `World::mAudioBus`) and a table of `SndBuf` instances (aka *Buffers*) that is used, for example, to hold sound data loaded from disk. An instance of `World` is accessible to `Unit` plug-ins via `Unit::mWorld` and provides `World::ft`, an instance of `InterfaceTable`, which is a table of function pointers which `Units` can invoke to perform operations on the `World`. An example of `Units` using `World` state is the `In` and `Out` units, which directly access `World::mAudioBus` to move audio data between `Graphs` and the global audio buses.

`Unit` subclasses provide all the signal-processing functionality of scsynth. They are defined in dynamically loaded executable "plug-ins." When the server starts, it scans the nominated plug-in directories and loads each plug-in, calling its `load()` function; this registers all available `Units` in the plug-in with the `World` via the `InterfaceTable::fDefineUnit` function pointer. Each call to `fDefineUnit()` results in a new `UnitDef` being created and registered with the global `gUnitDefLib` hash table, although this process is usually simplified by calling the macros defined in `SC_InterfaceTable.h`, such as `DefineSimpleUnit()` and `DefineDtorUnit()`.

Some server data (more of which we will see later) is kept away from `Unit` plug-ins in an instance of `HiddenWorld`. Of significance here are `HiddenWorld::mNodeLib`, a hash table providing fast lookup of `Nodes` by integer ID; `HiddenWorld::mGraphDefLib`, a hash table of all loaded `GraphDefs`, which is used when a request to instantiate a new `Graph` is received; and `HiddenWorld::mWireBufSpace`, which contains the memory used to pass data between `Units` during `Graph` evaluation.

30.3 Real-Time Implementation Structure

We now turn our attention to the context in which the server is executed. This includes considerations of threading, memory allocation, and interthread communications. Here, scsynth is a real-time system, and its implementation is significantly influenced by real-time requirements. We begin by considering what "real-time requirements" means in the context of scsynth and then explore how these requirements are met.

30.3.1 Real-Time Requirements

The primary responsibility of scsynth is to compute blocks of audio data in a timely manner in response to requests from the OS audio service. In general, the time taken to compute a block of audio must be less than the time it takes to play it. These blocks are relatively small (generally less than 2 milliseconds for current generation systems), and hence tolerances can be quite tight. Any delay in providing audio data to the OS will almost certainly result in an audible glitch.

Of course, computing complex synthesized audio does not come for free and necessarily takes time. Nonetheless, it is important that the time taken to compute each block is bounded and as close to constant as possible, so that exceeding timing constraints occurs only due to the complexity or quantity of concurrently active `Graphs`, not to the execution of real-time unsafe operations. Such unsafe operations include

- Algorithms with high or unpredictable computational complexity (for example, amortized time algorithms with poor worst-case performance)
- Algorithms which intermittently perform large computations (for example, precomputing a lookup table or zeroing a large memory block at `Unit` startup)
- Operations which block or otherwise cause a thread context switch

The third category includes not only explicit blocking operations, such as attempting to lock a mutex or wait on a file handle, but also operations which may block due to unknown implementation strategies, such as calling a system-level memory allocator or writing to a network socket. In general, any system call should be considered real-time unsafe, since there is no way to know whether it will acquire a lock or otherwise block the process.

Put simply, no real-time unsafe operation may be performed in the execution context which computes audio data in real time (usually a thread managed by the OS audio service). Considering the above constraints alongside the dynamic behavior implied by the domain model described in the previous section and the fact that scsynth can read and write sound files on disk, allocate large blocks of memory, and communicate with clients via network sockets, you may wonder how scsynth can work at all in real time. Read on, and all will be revealed.

30.3.2 Real-Time Messaging and Threading Implementation

SuperCollider carefully avoids performing operations which may violate real-time constraints by using a combination of the following techniques:

- Communication to and from the real-time context is mediated by lock-free First In First Out (FIFO) queues containing executable messages.
- Use of a fixed-pool memory allocator which is accessed only from the real-time context.
- Non-real-time safe operations (when they must be performed at all) are deferred and executed asynchronously in a separate "non-real-time" thread.
- Algorithms which could introduce unpredictable or transient high computational load are generally avoided.
- Use of user-configurable nonresizable data structures. Exhaustion of such data structures typically results in scsynth operations failing.

The first point is possibly the most important to grasp, since it defines the pervasive mechanism for synchronization and communication between non-real-time threads and the real-time context, which computes audio samples. When an non-real-time thread needs to perform an operation in the real-time context, it enqueues a message which is later performed in the real-time context. Conversely, if code in the real-time context needs to execute a real-time-unsafe operation, it sends the message to a non-real-time thread for execution. We will revisit this topic on a number of occasions throughout the remainder of the chapter.

Figure 30.2 shows another view of the scsynth implementation, this time focusing on the classes which support the real-time operation of the server. For clarity, only a few key classes from the domain model have been retained (shaded gray). Note that `AudioDriver` is a base class: in the implementation, different subclasses of `AudioDriver` are used depending on the target OS (`CoreAudioDriver` for Mac OS X, `PortAudioDriver` for Windows, etc.).

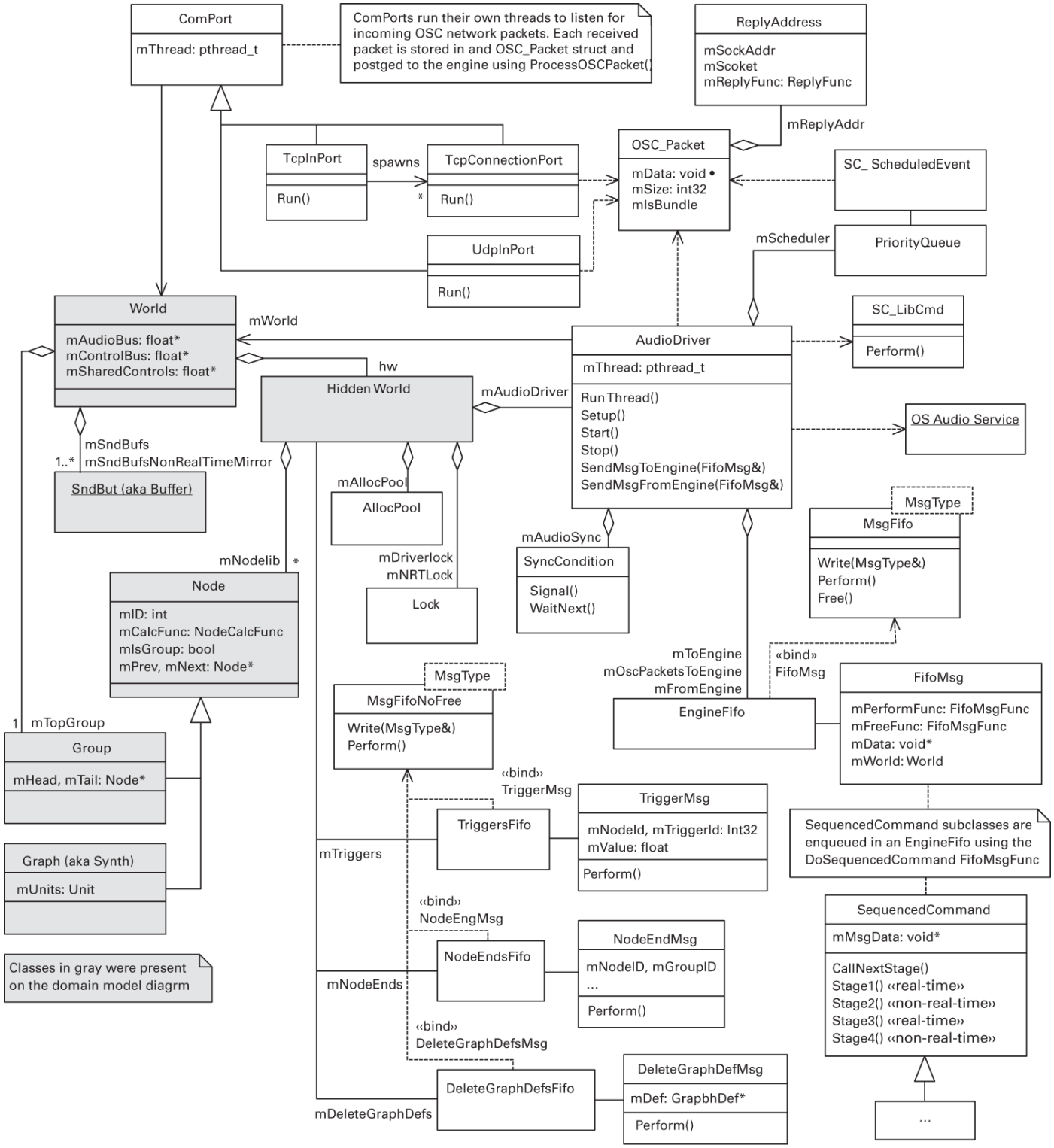


Figure 30.2
Real-time threading and messaging implementation structure.

Figure 30.3 illustrates the runtime thread structure and the dynamic communication pathways between threads via lock-free FIFO message queues. The diagram can be interpreted as follows: thick rectangles indicate execution contexts, which are either threads or callbacks from the operating system. Cylinders indicate FIFO message queue

objects. The padlock indicates a lock (mutex), and the black circle indicates a condition variable. Full arrows indicate synchronous function calls (invocation of queue-member functions), and half arrows indicate the flow of asynchronous messages across queues.

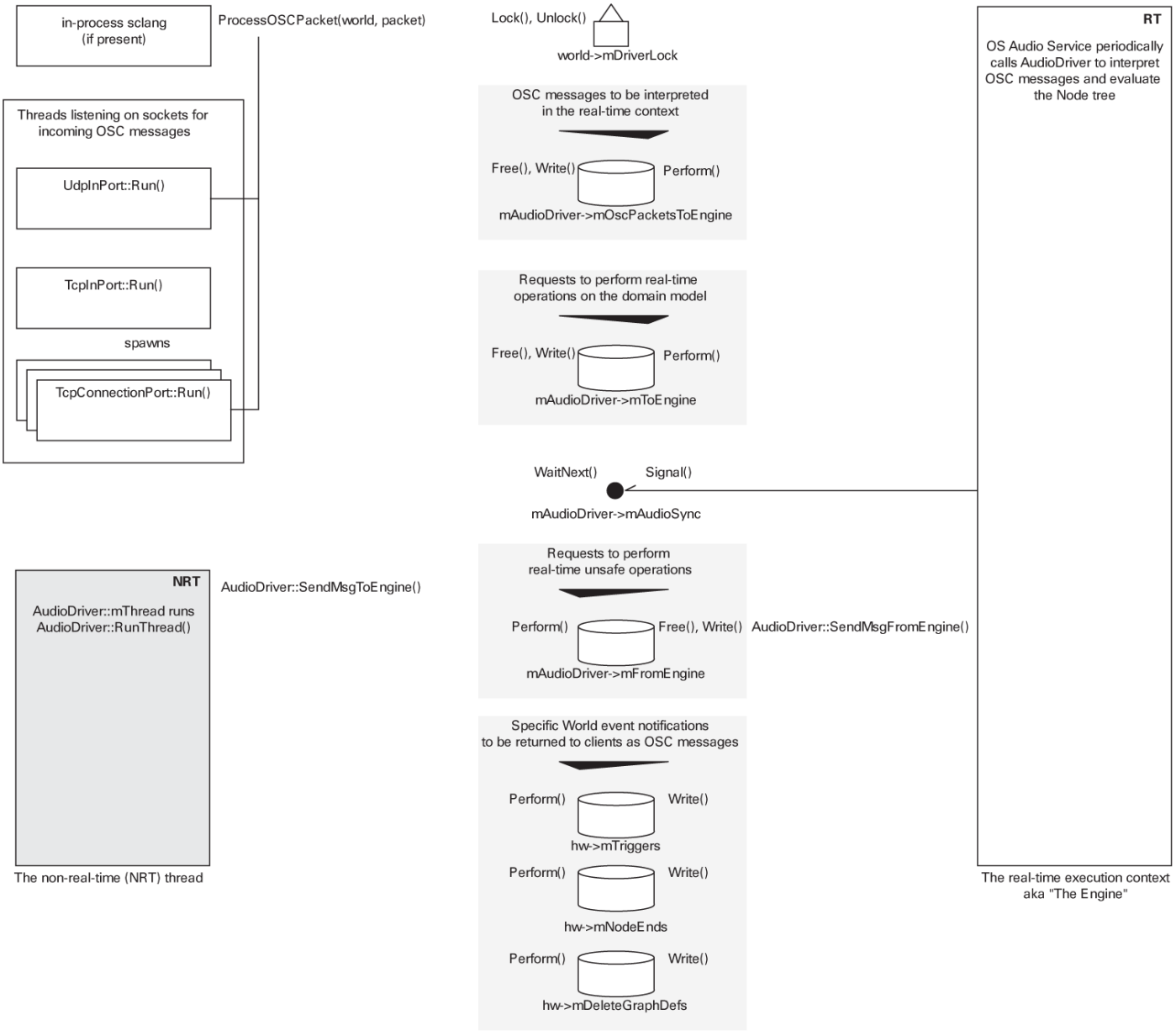


Figure 30.3
Real-time thread and queue instances and asynchronous message channels.

The FIFO message queue mechanism will be discussed in more detail later in the chapter, but for now, note that the `Write()` method enqueues a message, `Perform()` executes message-specific behavior for each pending message, and `Free()` cleans up after messages which have been performed. The `Write()`, `Perform()`, and `Free()` FIFO operations can be safely invoked by separate reader and writer threads without the use of locks.

With reference to figures 30.2 and 30.3, the dynamic behavior of the server can be summarized as follows:

1. One or more threads listen to network sockets to receive incoming OSC messages which contain commands for the server to process. These listening threads dynamically allocate instances of `OSC_Packet` and post them to "The Engine," using the `ProcessOSCPacket()` function, which results in `Perform_ToEngine_Msg()` (a `FifoMsgFunc`) being posted to the `mOscPacketsToEngine` queue. `OSC_Packet` instances are later freed, using `FreeOSCPacket()` (a `FifoFreeFunc`) by way of `MsgFifo::Free()`, via a mechanism which is described in more detail later.
2. "The Synthesis Engine," or "Engine" for short (also sometimes referred to here as "the real-time context"), is usually a callback function implemented by a concrete `AudioDriver` which is periodically called by the OS audio service to process and generate audio. The main steps relevant here are that the Engine calls `Perform()` on the `mOscPacketsToEngine` and `mToEngine` queues, which execute the `mPerformFunc` of any messages enqueued from other threads. Messages in `mOscPacketsToEngine` carry OSC_Packet instances which are interpreted to manipulate the `Node` tree, instantiate new `Graphs`, and so on. Whenever the Engine wants to perform a non-real-timesafe operation, it encodes the operation in a `FifoMessage` instance and posts it to the non-real-time thread for execution via the `mFromEngine` queue. Results of such operations (if any) will be returned via the `mToEngine` queue. After processing messages from `mOscPacketsToEngine`, `mToEngine` and any previously scheduled OSC messages in `mScheduler`, the Engine performs its audio duties by arranging for real-time audio data to be copied between OS buffers and `mWorld->mAudioBus` and evaluating the `Node` tree via `mWorld->mTopGroup`. When the Engine has completed filling the OS audio output buffers, it calls `Signal()` on `mAudioSync` and returns to the OS.
3. Before the server starts servicing OS audio requests, it creates a thread for executing real-time-unsafe operations (the NRT or NRT thread). This thread waits on `mAudioSync` until it is signaled by the Engine. When the NRT thread wakes up, it calls `Free()` and `Perform()` on the `mFromEngine` queue to perform any NRT-safe operations which the server has posted, and then processes the `mTriggers`, `mNodeEnds`, and `mDeleteGraphDefs` queues. These queues contain notifications of server events. Performing the enqueued notification messages results in OSC messages being sent to clients referenced by `ReplyAddress`. After calling `Perform()` on all queues, the NRT thread returns to waiting for `mAudioSync` until it is next wakened by the Engine. Note that `mAudioSync` is used to ensure that the NRT thread will always wake up and process Engine requests in a timely manner. However, it may never sleep, or it may not process the queues on every Engine cycle if it is occupied with time-consuming

operations. This is acceptable since the Engine assumes that NRT operations will take as long as necessary.

The description above has painted the broad strokes of the server's real-time behavior. Zooming in to a finer level of detail reveals many interesting mechanisms which are worth the effort to explore. A number of these are discussed in the sections which follow.

30.3.3 Real-Time Memory Pool Allocator

Memory allocations performed in the real-time context, such as allocating memory for new `Graph` instances, are made using the `AllocPool` class. `AllocPool` is a reimplementation of Doug Lea's fast general-purpose memory allocator algorithm (Lea, 2000). The implementation allocates memory to clients from a large, preallocated chunk of system memory. Because `AllocPool` is invoked only by code running in the real-time context, it doesn't need to use locks or other mechansims to protect its state from concurrent access and hence is real-time safe. This makes it possible for the server to perform many dynamic operations in the real-time thread without needing to defer to an NRT thread to allocate memory. That said, large allocations and other memory operations which are not time critical are performed outside the real-time context. Memory allocated with an `AllocPool` must of course also be freed into the same `AllocPool`, and in the same execution context, which requires some care to be taken. For example, `FifoMsg` instances posted by the Engine to the NRT thread with a payload allocated by `AllocPool` must ensure that the payload is always freed into `AllocPool` in the real-time execution context. This can be achieved using `MsgFifo::Free()`, which is described in the next section.

30.3.3.1 FIFO queue message passing As already mentioned, scsynth uses FIFO queues for communicating between threads. The basic concept of a FIFO queue is that you push items on one end of the queue and pop them off the other end later, possibly in a different thread. A fixed-size queue can be implemented as a *circular buffer* (also known as a *ring buffer*) with a read pointer and a write pointer: new data are placed in the queue at the write pointer, which is then advanced; when the reader detects that the queue is not empty, data are read at the read pointer and the read pointer is advanced. If there's guaranteed to be only 1 reading thread and 1 writing thread, and you're careful about how the pointers are updated (and take care of atomicity and memory-ordering issues), then it's possible to implement a thread-safe FIFO queue without needing to use any locks. This lock-free property makes the FIFO queue ideal for implementing real-time interthread communications in scsynth.

The queues which we are most concerned with here carry a payload of message objects between threads. This is an instance of the relatively well known *Command*

design pattern (Gamma et al., 1995). The basic idea is to encode an operation to be performed as a class or struct and then pass it off to some other part of the system for execution. In our case, the `Command` is a struct containing data and a pair of function pointers, one for performing the operation and another for cleaning up. We will see later that scsynth also uses a variant of this scheme in which the `Command` is a C++ class with virtual functions for performing an operation in multiple stages. But for now, let's consider the basic mechanism, which involves posting `FifoMsg` instances to a queue of type `MsgFifo`.

Figure 30.2 shows that `mOscPacketsToEngine`, `mToEngine`, and `mFromEngine` queues carry `FifoMsg` objects. The code below shows the `FifoMsgFunc` type and the key fields of `FifoMsg`:

```
typedef void (*FifoMsgFunc)(struct FifoMsg*);
struct FifoMsg {
. . .
FifoMsgFunc mPerformFunc;
FifoMsgFunc mFreeFunc;
void* mData;
. . .
};
```

To enqueue a message, the sender initializes a `FifoMsg` instance and passes it to `MsgFifo::Write()`. Each `FifoMsg` contains the function pointer members `mPerformFunc` and `mFreeFunc`. When the receiver calls `MsgFifo::Perform()`, the `mPerformFunc` of each enqueued message is called with a pointer to the message as a parameter. `MsgFifo` also maintains an additional internal pointer, which keeps track of which messages have been performed by the receiver. When `MsgFifo::Free()` is called by the sending execution context, the `mFreeFunc` is invoked on each message whose `mPerformFunc` has already completed. In a moment, we will see how this mechanism is used to free `SequencedCommand` objects allocated in the real-time context.

A separate `MsgFifoNoFree` class is provided for those FIFOs which don't require this freeing mechanism, such as `mTriggers`, `mNodeEnds`, and `mDeleteGraphDefs`. These queues carry specialized notification messages. The functionality of these queues could have been implemented by dynamically allocating payload data and sending it using `FifoMsg` instances; however, since `MsgFifo` and `MsgFifoNoFree` are templates parameterized by message type, it was probably considered more efficient to create separate specialized queues using message types large enough to hold all of the necessary data rather than invoking the allocator for each request.

The FifoMsg mechanism is used extensively in scsynth, not only for transporting OSC message packets to the real-time engine, but also for arranging for the execution of real-time-unsafe operations in the NRT thread. Many server operations are implemented by the FifoMsgFuncs defined in *SC_MiscCmds.cpp*. However, a number of operations need to perform a sequence of steps alternating between the real-time context and the NRT thread. For this, the basic FifoMsg mechanism is extended using the SequencedCommand class.

30.3.4 SequencedCommand

Unlike FifoMsg, which just stores two C function pointers, SequencedCommand is a C++ abstract base class with virtual functions for executing up to 4 stages of a process. Stages 1 and 3 execute in the real-time context, while stages 2 and 4 execute in the NRT context. The Delete() function is always called in the RT context, potentially providing a fifth stage of execution. SequencedCommands are used for operations which need to perform some of their processing in the NRT context. At the time of writing, all SequencedCommand subclasses were defined in *SC_SequencedCommand.cpp*. They are mostly concerned with the manipulation of SndBufs and GraphDefs. (See table 30.1 for a list of SequencedCommands defined at the time of writing.)

Table 30.1
Subclasses of SequencedCommand Defined in *SC_SequencedCommand.cpp*

Buffer commands	BufGenCmd, BufAllocCmd, BufFreeCmd, BufCloseCmd, BufZeroCmd, BufAllocReadCmd, BufAllocReadChannelCmd, BufReadCmd, BufReadChannelCmd, SC_BufReadCommand, BufWriteCmd
GraphDef commands	LoadSynthDefCmd, RecvSynthDefCmd, LoadSynthDefDirCmd
Miscellaneous	AudioQuitCmd, AudioStatusCmd, SyncCmd, NotifyCmd, SendFailureCmd, SendReplyCmd, AsyncPlugInCmd

To provide a concrete example of the SequencedCommand mechanism, we turn to the Help file for Buffer (aka SndBuf), which reads: “Buffers are stored in a single global array indexed by integers beginning with zero. Buffers may be safely allocated, loaded and freed while synthesis is running, even while unit generators are using them.” Given that a SndBuf’s sample storage can be quite large, or contain sample data read from disk, it is clear that it needs to be allocated and initialized in the NRT thread. We now describe how the SequencedCommand mechanism is used to implement this behavior.

To begin, it is important to note that the SndBuf class is a relatively lightweight data structure which mainly contains metadata such as the sample rate, channel count, and number of frames of the stored audio data. The actual sample data are stored in a dynamically allocated floating-point array pointed to by SndBuf::data. In the

explanation which follows, we draw a distinction between instance data of `SndBuf` and the sample data array pointed to by `SndBuf::data`.

In contrast to the client-oriented worldview presented in the Help file, `World` actually maintains 2 separate arrays of `SndBuf` instances: `mSndBufs` and `mSndBufsNonRealTimeMirror`. Each is always in a consistent state but is accessed or modified only in its own context: `mSndBufs` in the RT context via `World_GetBuf()` and `mSndBufsNonRealTimeMirror` in the NRT thread via `World_GetNRTBuf()`. On each iteration, the engine performs messages in `mToEngine` and then evaluates the `Node` tree to generate sound. Any changes to `mSndBufs` made when calling `mToEngine->Perform()` are picked up by dependent `Units` when their `UnitCalcFunc` is called.

The code may reallocate an existing `SndBuf`'s sample data array. It is important that the old sample data array is not freed until we can be certain no `Unit` is using it. This is achieved by deferring freeing the old sample data array until after the new one is installed into the `RT` context's `mSndBufs` array. This process is summarized in figure 30.4. The details of the individual steps are described below.

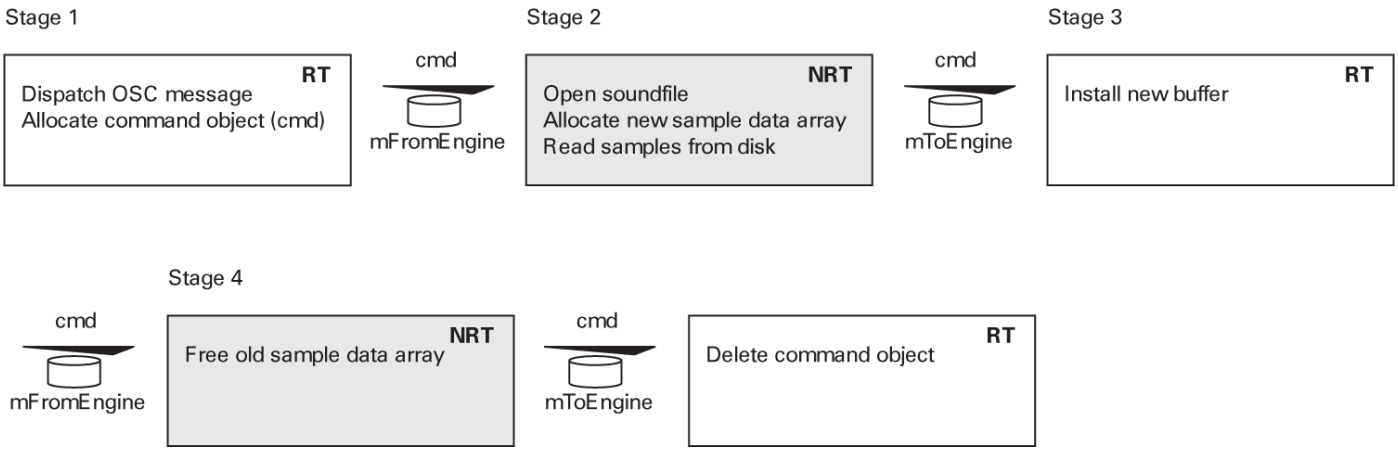


Figure 30.4
Overview of multithreaded processing of the `/b_allocRead` command.

We now consider the steps performed at each stage of the execution of `BufAllocReadCmd`, a subclass of `SequencedCommand`, beginning with the arrival of an `OSC_Packet` in the real-time context. These stages are depicted in four sequence diagrams, displayed in figures 30.5–30.8. The exact function parameters have been simplified from those in the source code, and only the main code paths are indicated to aid understanding. The OSC message to request allocation of a `Buffer` filled with data from a sound file is as follows:

```
/b_allocRead bufnum path startFrame numFrames
```

Stage 1 (see figure 30.5): The real-time context processes an OSC packet containing the `/b_allocRead` message. The OSC dispatch mechanism looks up the correct function pointer to invoke from `gCmdLib`, in this case `meth_b_allocRead()`.`meth_b_allocRead()` calls `CallSequencedCommand()` to instantiate a new `BufAllocReadCmd` instance (a subclass of `SequencedCommand`) which we will call `cmd`. `CallSequencedCommand()` calls `cmd->Init()`, which unpacks the parameters from the OSC packet and then calls `cmd->CallNextStage()`, which in turn invokes `cmd->Stage1()`, which in the case of `BufAllocReadCmd` does nothing. It then enqueues `cmd` to the NRT thread, using `SendMessageFromEngine()` with `DoSequencedCommand` as the `FifoMsgFunc`.

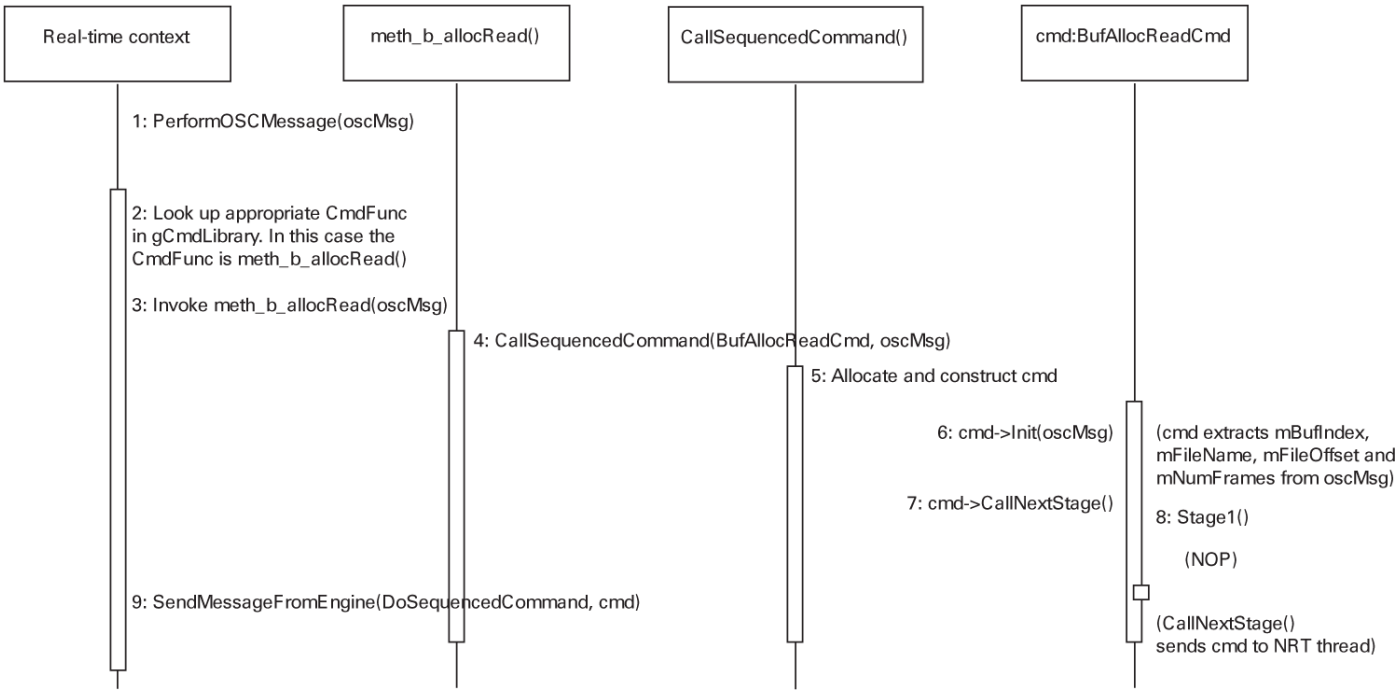


Figure 30.5
Stage 1 of processing the `/b_allocRead` command in the real-time context.

Stage 2 (see figure 30.6): Some time later, the `mFromEngine` FIFO is processed in the NRT thread. The `FifoMsg` containing our `cmd` is processed, which results in `cmd->Stage2()` being called via `DoSequencedCommand()` and `cmd->CallNextStage()`. `cmd->Stage2()` does most of the work: first, it calls `World_GetNRTBuf()`, which retrieves a pointer to the NRT copy of the `SndBuf` record for `cmd->mBufIndex`. Then it opens the sound file and seeks to the appropriate position. Assuming no errors have occurred, the pointer to the old sample data array is saved in `cmd->mFreeData` so it can be freed later. Then `allocBuf()` is called to update the `SndBuf` with the new file information and to allocate a new sample data array. The data are read from the file into the sample data array, and the file is closed. A shallow copy of the NRT `SndBuf` is

saved in `cmd->mSndBuf`. Finally, `cmd->CallNextStage()` enqueues the `cmd` with the real-time context.

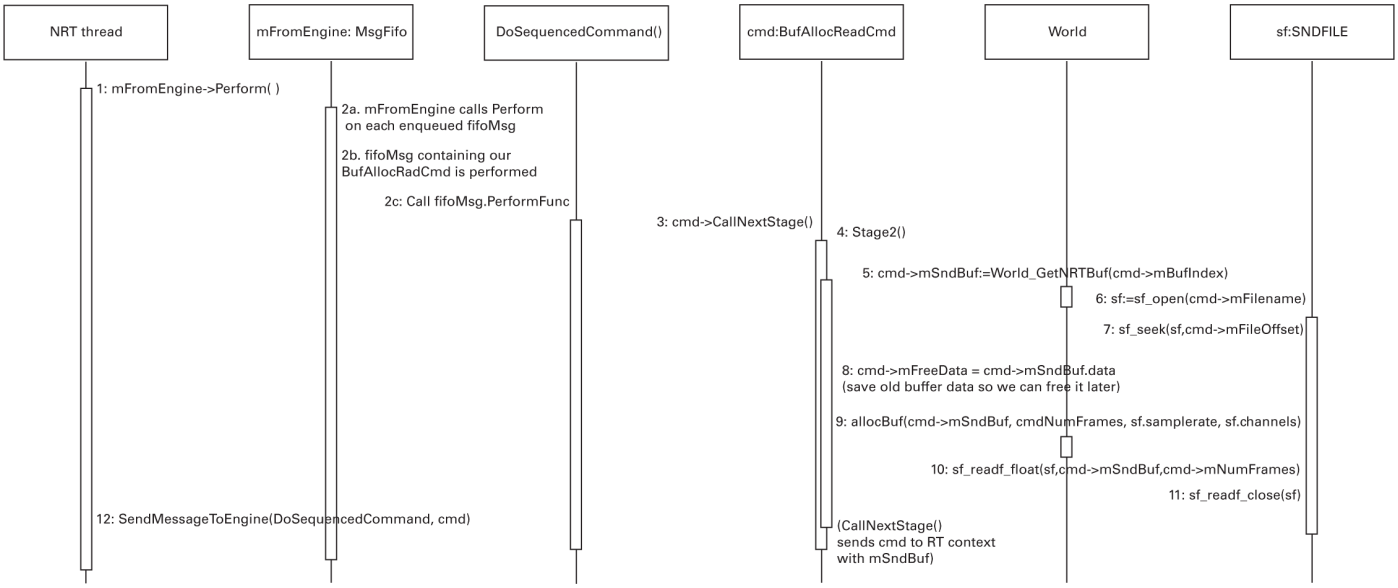


Figure 30.6
Stage 2 of processing the `/b_allocRead` command in the NRT context.

Stage 3 (see figure 30.7): Similarly to stage 2, only this time in the real-time context, `cmd->Stage3()` is called via `DoSequencedCommand()` and `cmd->CallNextStage()`. A pointer to the *real-time* copy of the `SndBuf` for index `cmd->mBufIndex` is retrieved using `World_GetBuf(cmd->mBufIndex)`, and the `SndBuf` instance data initialized in stage 2 is shallow-copied into it from `cmd->mSndBuf`. At this stage, the sample data array which was allocated and loaded in stage 2 is now available to `Units` calling `World_GetBuf().cmd` is then sent back to the non-real-time thread.

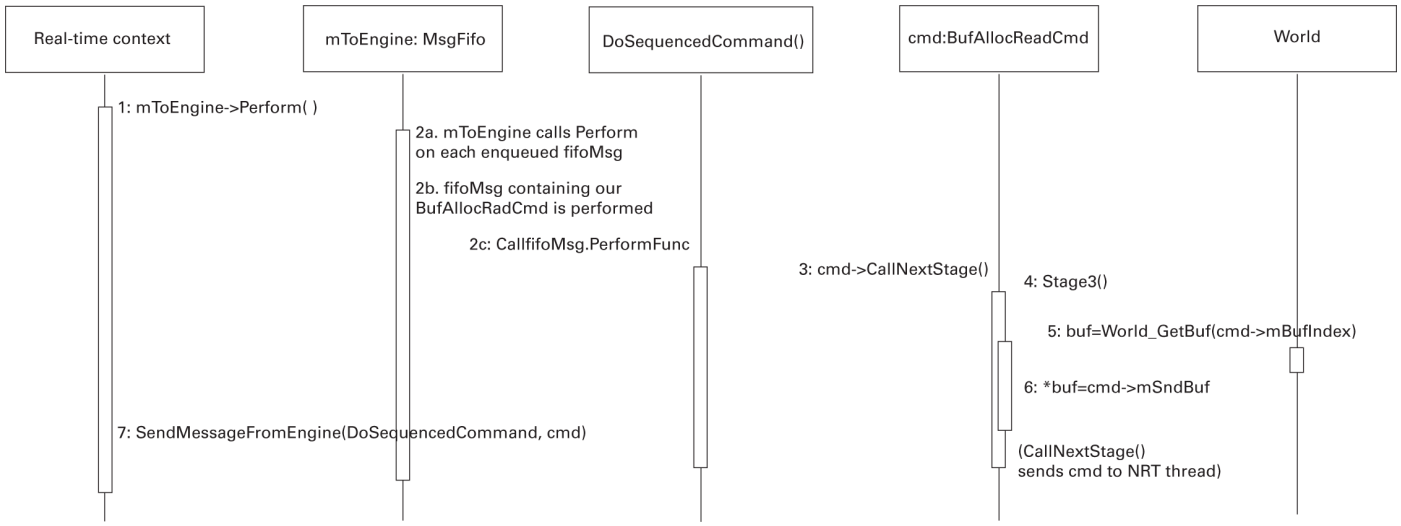


Figure 30.7
Stage 3 of processing the `/b_allocRead` command in the real-time context.

Stage 4 (see figure 30.8): Once again, back in the non-real-time thread, `cmd->Stage4()` is invoked, which frees the old sample data array which was stored into `cmd->mFreeData` in stage 2. Then the `SendDoneWithIntValue()` routine is invoked, which sends an OSC notification message back to the client that initiated the `Buffer` allocation. Finally, `cmd` is enqueued back to the real-time context with the `FreeSequencedCommand FifoMsgFunc`, which will cause `cmd` to be freed, returning its memory to the real-time `AllocPool`.

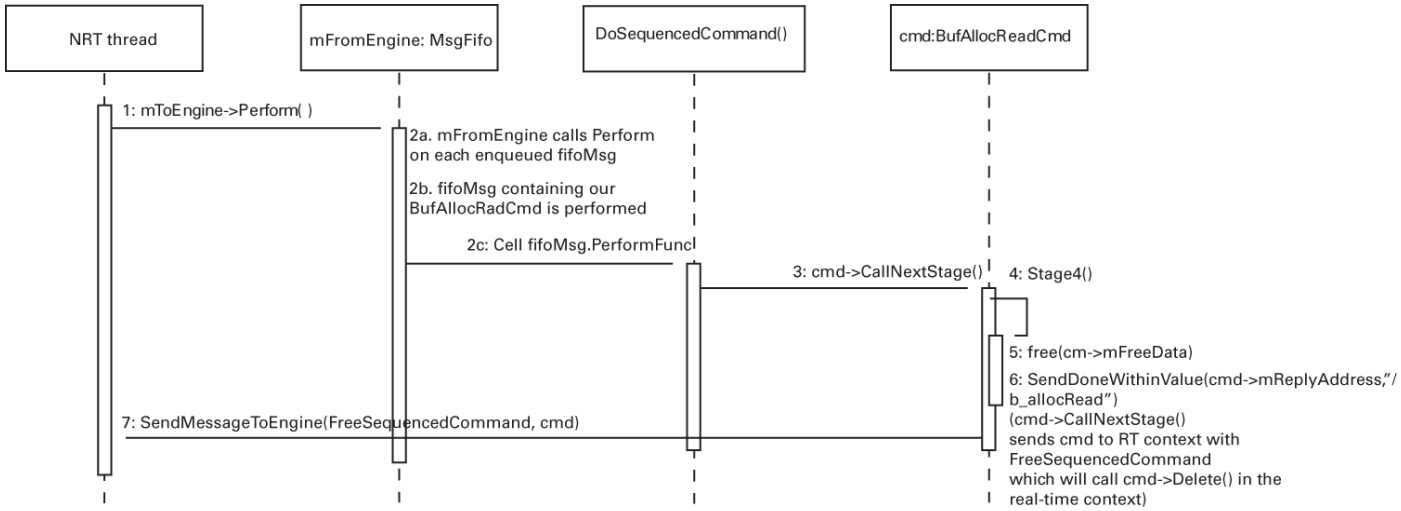


Figure 30.8
Stage 4 of processing the `/b_allocRead` command in the NRT (NRT) context.

30.3.5 Processing and Dispatching OSC Messages

The `ProcessOSCPacket()` function provides a mechanism for injecting OSC messages into the real-time context for execution. It makes use of `mDriverLock` to ensure that only one thread is writing to the `mOscPacketsToEngine` queue at any time (this could occur, for example, when multiple socket listeners are active). To inject an OSC packet using `ProcessOSCPacket()`, the caller allocates a memory block using `malloc()`, fills it with an OSC packet (for example, by reading from a network socket), and then calls `ProcessOSCPacket()`.`ProcessOSCPacket()` takes care of enqueuing the packet to the `mOscPacketsToEngine` queue and deleting packets, using `free()`, once they are no longer needed.

Once the real-time context processes OSC packets, they are usually freed using the `MsgFifo` message-freeing mechanism; however, packets whose time-stamp values are in the future are stored in the `mScheduler PriorityQueue` for later execution. Once a scheduled packet has been processed, it is sent to the NRT thread to be freed.

scsynth dispatches OSC commands by looking up the `SC_CommandFunc` associated with a given OSC address pattern. At startup, *SC_MiscCmds.cpp* wraps these functions in `LibCmd` objects and stores them in both the `gCmdLib` hash table and `gCmdArray` array.

OSC commands sent to the server may be strings or special OSC messages with a 4-byte address pattern in which the low byte is an integer message index. Command strings are compatible with any OSC client, whereas the integer command indices are more efficient but don't strictly conform to the OSC specification. When integer command indices are received, `PerformOSCMessage()` looks up the appropriate `SC_CommandFunc` in the `gCmdArray` array; otherwise, it consults the `gCmdLib` hash table.

The `mTriggers`, `mNodeEnds`, and `mDeleteGraphDefs` FIFOs are used by the real-time context to enqueue notifications which are translated into OSC messages in the NRT thread and are sent to the appropriate reply address by invoking `ReplyAddress::mReplyFunc`.

30.3.6 Fixed-Size Data Structures

In real-time systems, a common way to avoid the potential real-time-unsafe operation of reallocating memory (which may include the cost of making the allocation and of copying all of the data) is simply to allocate a "large enough" block of memory in the first place and have operations fail if no more space is available. This fixed-size allocation strategy is adopted in a number of places in scsynth, including the size of

- FIFO queues which interconnect different threads `mAllocPool` (the real-time context's memory allocator)
- The `mScheduler` priority queue for scheduling OSC packets into the future
- The `mNodeLib` hash table, which is used to map integer `Node` IDs to `Node` pointers

In the case of `mNodeLib`, the size of the table determines the maximum number of `Nodes` that the server can accommodate and the speed of `Node` lookup as `mNodeLib` becomes full. The sizes of many of these fixed-size data structures are configurable in `WorldOptions` (in general, by command-line parameters), the idea being that the default values are usually sufficient, but if your usage of scsynth causes any of the default limits to be exceeded, you can relaunch the server with larger sizes as necessary.

30.4 Low-Level Mechanisms

As may already be apparent, scsynth gains much of its power from efficient implementation mechanisms. Some of these fall into the category of low-bounded complexity methods which contribute to the real-time capabilities of the server, while others are more like clever optimizations which help the server to run faster. Of course, the whole server is implemented efficiently, so looking at the source code will reveal many more optimizations than can be discussed here; however, a number of those which I have found interesting are briefly noted below. As always, consult the source code for more details:

- The "str4" functions use a string that is 4-byte aligned and zero-padded, stored as an array of 32-bit integers. Aside from being the same format that OSC uses, the implementation improves the efficiency of comparison and other string operations by being able to process 4 chars at once.
- Hash tables in scsynth are implemented using open addressing with linear probing for collision resolution. Although these tables don't guarantee constant time performance in the worst case, when combined with a good hashing function (Wang, 2007), they typically provide close to constant performance, so long as they don't get too full.
- One optimization to hashing used in a number of places in the source code is that the hash value for each item (such as a `Node`) is cached in the item. This improves performance when resolving collisions during item lookup.
- The `World` uses a "touched" mechanism which Units and the `AudioDriver` can use to determine whether audio or control buses have been filled during a control cycle: `World` maintains the `mBufCounter`, which is incremented at each control cycle. When a Unit writes to a bus, it sets the corresponding touched field (for example, in the `mAudioBusTouched` array for audio buses) to `mBufCounter`. Readers can then check the touched field to determine whether the bus contains data from the current control cycle. If not, the data doesn't need to be copied and zeros can be used instead.
- Delay lines typically output zeros until the delay time reaches the first input sample. One way to handle this is to zero the internal delay storage when the delay is created or reset. The delay unit generators in scsynth (see *DelayUGens.cpp*) avoid this time-

consuming (and hence real-time unsafe) operation by using a separate `UnitCalcFunc` during the startup phase. For example, `BufDelayN_next_z()` outputs zeros for the first `bufSamples` samples, at which point the `UnitCalcFunc` is switched to `BufDelayN_next()`, which outputs the usual delayed samples.

• For rate-polymorphic units, the dynamic nature of UnitCalcFuncs is used to select functions specialized to the rate type of the Unit's parameters. For example, `BinaryOpUgens.cpp` defines UnitCalcFuncs which implement all binary operations in separate versions for each rate type. For example, there are separate functions for adding an audio vector to a constant, `add_ai()`, and adding two audio vectors, `add_aa()`. When the binary-op Unit constructor `BinaryOpUGen_Ctor()` is called, it calls `ChooseOperatorFunc()` to select among the available UnitCalcFuncs based on the rate of its inputs.

This concludes our little journey through the wonderful gem that is scsynth. I invite you to explore the source code yourself; it has much to offer, and it's free!

References

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Design*. Reading, MA: Addison-Wesley.

Lea, D. 2000. "A Memory Allocator." Accessed January 9, 2008. http://g.oswego.edu/dl/html/malloc.html.

McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal,* 26(4): 61–68.

Wang, T. 1997. "Integer Hash Function." Accessed January 9, 2008, from http://www.concentric.net/~Ttwang/tech/inthash.htm.

Subject Index

This index includes topics from the main body of the text. Ubiquitous topics have been limited to principal references. For messages and classes from the SC language, see the code index. For definitions of terms, see the syntax appendix.

12-Tone Matrix, 39

Abstraction, 208–209. *See also chapter 7*
Additive Synthesis, 9, 12, 40–42, 130
AIFF, 10, 194, 249
Algorithm (algorithmic), 154, 421, 530
 composition, 613
 inside a method, 161–162
 synthesis, 359, 387, 392, 533, 626, , 634
Ambisonics, 391–392
Analysis
 FFT, 396
 real time, 404
 UGens, 117
Arduino, 116–117
Arguments, 11, 13–15, 133–135, 148–150
Array, 15, 19, 20–21, 35–37, 39, 41, 53–54
 indexing, 234–236
 nested, 89
ASCII, 112, 130, 165
Association, 163
Audio rate, 24, 48, 58, 195

Balancing enclosures, 12
Beat Tracking. *See* Machine listening
Binary, 66, 135–136
 numbers, 622, 624
 operators, 18
Binaural, 387–389
Bipolar, 27, 48–50, 59, 66
Boolean, 36, 66
BPF (Band pass filter). *See* Filter
Buffer, 31–35, 62, 77, 90–93, 152, 184, 198, 203, 349, 383–386, 410–414, 439–444, 455–458, 468, 506–510, 558–569, 665–672, 714–715, 753, 764–767, 773–806
Bus, 28–34, 59–62, 73–74, 80, 180–185, 202–203, 210–213, 226–227, 416–417, 503–510, 517–519, 524–529, 557–568, 779, 793, 798
Byte Code, 148–149, 719–720, 726–743,

C++, 57, 117, 130, 177, 235, 307, 442, 642, 673, 719–720, 752–767, 776–793, 801–802
Carrier (phase modulation), 22–23, 28, 47, 56
Cents, 470–472, 484
Char, 131–132, 162, 328–329, 577, 581–582, 722–728, 739, 744, 746–748
Class (classes), 58, 140–142, 150, 161–173, 179–190

as object models, 237–239
tree, 173
writing, 763–764
Clock (class), 83, 87, 216, 224
AppClock, 83, 329, 602, 607, 680–681
SystemClock, 68, 82–84, 329, 509, 607
TempoClock, 83–87, 185, 197–200, 584–592, 607, 627–628
Cloud (CloudGenerator), 251, 257–258, 437–438
Coding
conventions, 719
networked live, 229, 359, 642
scsynth style, 792
Collection, 15, 17, 20, 25, 35, 41, 112, 115 125, 130–136, 152, 162–165
Comb (N, L, C), 16, 49, 77, 78, 383–384
Combinatorics, 226
Comments, 11, 17, 54, 241, 578, 625, 785
Compilation (compiler), 147, 725–730, 750
Composition. *See chapter 3*
DAW style, 55
object oriented (*see chapter 18*)
Compression, 63
Conductor. *See* Patterns, conductor
Constraints. *See chapter 23*
Control rate, 24, 32, 47, 72–79, 195, 212, 227–228
ControlSpec, 124–125, 324–329, 335, 680
Convolution, 385–389, 446, 464
CPU (usage), 22, 73–77, 94, 257, 267–268, 367, 397, 785–786
Crucial library. *See* Libraries, crucial
Csound, 464, 794
Cue Players, 90

DAW (Digital audio workstation) Composition (*see* Composition, DAW)
DC (offset), 214
Debugging, 52, 57, 64, 119, 578, 593, 783
Decorrelation, 395–397, 401, 562
Delay, 34–35, 53–54, 63–68, 77–83, 383–385, 395,
Devices, external. *See* External devices
Dialects, 617–637. *See also chapter 23*
Dialogue (windows), 152
Dictionary, 100, 140, 143, 162–166, 183, 200–201, 249–250, 439, 464, 729–730
Distortion, 63, 295, 307, 454, 458, 460, 552, 563, 565
Document, Emacs, 265
Dot (receiver dot message), 16–17, 17, 27, 136
Drag and Drop, 266

Emacs (scel), 265–266, 278–293
Encapsulation, 517–526, 701, 792
Enclosures, 18, 52
Envelope, 24–27, 40, 63, 74–75, 98, 195, 335, 423, 426–455, 541, 600–607
Environment, 57, 140–143, 163, 166–167, 629
variables, 31, 46, 140
Evaluation (of code), 10, 14, 147
Event, 182–184. *See also* Patterns; *chapter 6*

note (keys), 184–189
as object models, 237–239
PatternProxy, 212, 217–218
protoEvent, 192, 194, 200–203
streams, 87, 100, 199, 218, 224, 230, 470, 530
triggering, 57
Extensions. *See* Libraries
External devices. *See chapter 4*

FFT, 396–399, 404–406, 506,
Filter, 12, 24, 69, 80, 93
BPF, 117, 210, 414, 452, 592
HPF, 80, 117, 349, 414, 556, 759
Klank, 43, 53, 780
Lag (Lag2), 63, 70, 185, 209
LeakDC, 368, 553, 555–561
LPF, 80, 117, 553, 556–568,
Median, 117
Ringz, 226–227, 362
RLPF, 12, 710–711, 714–716
Flange, 757–773
Float (floating-point), 14, 17, 59, 69
Flow control, 159–160
FlowLayout, 96, 159, 258–260, 478
Fourier, 385, 423
Frequency modulation (FM). *See* Modulation, frequency
FreeVerb, 69, 351, 386, 661
Function, 14, 17–20, 36, 61, 144–145, 148
FunctionDef, 142, 149, 732, 744
iterating, 35, 152, 154, 569,
return, 132, 145

Garbage collection, 719–720, 743, 745, 751
Gate, 24, 65, 69–78, 90–92, 179–181, 184
Gestures, 96
Granular synthesis, 64, 80, 195–196, 251–253, 397, 424, 438, 450, 465, 548. *See also* Microsound
client-side, 432, 488
server side, 471, 483
sound files, 488–489
wave sets, 490–500
Grouping. *See* Precedence
GUI (Graphical user interface). *See also* Platforms; *chapters 9–12*
cross-platform, 266, 276
dynamically generated, 320
Emacs (*see chapter 12*)
OS X, 83
tuning, 478
GVerb, 69–71, 386–387, 639

Harmonic spectrum. *See* Spectrum, harmonic series
HID (Human Interface Devices), 103–108, 113, 118–119,
Linux, 118
HierSch, 626–629

History, 229, 232, 239
HPF (High pass filter). *See* Filter
Human Interface Devices. *See* HID

IdentityDictionary, 162, 165–166, 183, 200, 235
If (statements). *See* Flow control
Inharmonic spectrum. *See* Spectrum, inharmonic
Inheritance, 131, 168–171
Instance methods, 132, 139–143, 159, 169, 172
Instance variables. *See* Variables, instance
Interpolation, 58, 66, 69, 73, 77, 78–80, 224, 325, 383, 431, 456, 460
Interpreter, 180, 183, 203, 206, 241, 248, 305, 740–750
  variables, 141–142
Introspection. *See* Linux, introspection
iPhone, 295, 570, 615
Iteration, 35–36, 44–47, 152–154
ixiQuarks, 596–597, 603–609

JACK. *See* Linux, JACK
Japan. *See* *chapter 22*
Java, 130–288, 593, 625, 637, 673
JITLib (Just In Time), 228–229, 629, 645
JSCUserView. *See* GUI
Juggling, 372

Key Tracking. *See* Machine listening
Keyboard and Mouse, 311
Keywords, 22–23, 134, 171, 574, 593
Klank. *See* Filter

Lag (Lag2), 63, 70, 114, 121, 185, 669
LazyEnvir, 209, 212, 229, 629–630
LeakDC. *See* Filter Libraries
Linear. *See* Interpolation
Linux, 110, 118, 125, 276, 278. *See also* *chapter 11*
  ALSA, 298–308
  JACK, 298, 301, 303
Live performance. *See* *chapter 20*
ListPattern, 142
Literals, 130–132, 574, 577–579
Localization, 353
Logical expressions, 35, 38, 47. *See also* ==, !=, >, <, >=, <=, and, or *in code index*
Loop, 31–35
  infinite, 153, 216
LPF (Low pass filter). *See* Filter

Mac OS X. *See* Platforms
Machine listening. *See* *chapter 15*
  beat tracking, 413–414
  key tracking, 415
  onset detection, 415
  transcription, 415, 417
Map (Mapping), 34–35, 50, 60, 63, 66, 103, 106, 121–124, 214, 234, 284–288, 323, 326, 347–365, 367, 373, 476, 483–484, 511–517, 555–558, 598–608, 620–624, 647, 651, 659, 662–664, 669–676, 687–688

Markov, 47, 645, 648, 659–662
Matrix, 12-Tone. *See* 12-Tone matrix
Max/MSP, 95, 418, 650, 720
Median. *See* Filter
Message (method), 102, 132–134
 chains, 132
 instance, 131
 nested, 15, 16
Method. *See* Message
MetaClass, 172
Microsound. *See* Granular synthesis; *chapter 16*
MIDI, 29–30, 35–38, 67–70, 78–79, 84–89, 105, 107–125, 328, 468, 470–473, 536, 604
MIDIIn, 107–111, 329, 330, 419
MIDIResponder, 329
Modulation, 79
 frequency, 21, 23, 79, 226 353, 439, 446, 712
 index, 23
 phase, 9, 21–23, 26, 28, 558, 669, 670, 709, 711–712
 pulse-width, 117
Modulo (Mod, %), 38, 206, 223–224, 552, 556
Mono (Monophonic), 21, 31, 60, 69, 74, 91–92, 380, 384, 441, 554, 598
Mouse. *See* Keyboard and Mouse
Multichannel (expansion), 20–24, 59–60, 203, 379–382, 634, 757, 779–780

Nesting, 15–18, 22, 55, 87, 101
Networked live coding. *See* Coding, networked live
Nil, 53, 100–101, 112–115, 141, 240–241, 683
Node, 31, 60–62, 170, 184–185, 208–213, 219, 226–230, 430, 439
NodeProxy, 209, 210, 213, 224, 226, 228–229, 370, 439
Noise, 58, 63, 68, 295, 307, 651–652
Nyquist, 406, 782–783

Object(s), 129
 layout, 720
 modeling (*see chapter 8*)
 oriented composition (*see chapter 18*)
Offset, 47–50
Onset detection. *See* Machine listening
Open Sound Control. *See* OSC
Operators, 18, 36, 135–136, 163
Optimization, 57, 62, 73–74, 315, 373, 784–785, 809
OS X. *See* Platforms
OSC (Open Sound Control), 66, 89, 103, 111, 181, 673, 685

Panning, 59, 63, 73, 98, 353, 380–382, 400, 439, 446, 545–547
Patterns, 185–227. *See also* Events; *chapter 6*
Phase, 32, 58, 63, 130, 232
 modulation (*see* Modulation, phase)
 spectrum decorrelation, 396
Physical Model, 9, 12, 43, 533, 626
Pink Noise, 77–78, 209, 332–324, 357 , 368, 391, 399, 592, 709
Platforms
 Linux (*see chapter 12*)

Max OS X (*see chapter 9*)
Windows (*see chapter 11*)
Plug-ins. *See chapter 25*
Polymorphism, 88, 89, 168, 233–234, 526–529, 619, 720
Precedence, 18–19, 42, 135–136, 381, 400, 709
Precedence effect, 381–382, 400
Programming. *See chapter 5*
Primitives, 170, 710–720, 726, 740, 745–750
Prototypes, 352, 622
ProxySpace, 207–210, 226, 228–229, 236, 426, 439–444, 629, 639, 644, 648

QCD (quantum chromodynamics), 248–250, 347, 371–372
Quantization, 197, 224–226
Quarks, 103, 111, 117, 125, 176, 249, 283, 408, 438, 600, 634, 790. *See also* ixiQuarks

Random
number generators, 14, 524, 625, 727, 775
patterns, 624
parameter choices, 639
pseudo, 514, 622–624, 775
range, 134, 251, 254, 434, 437–438 (*see also* rrand *in code index*)
seed, 14, 70
server side 68–69
Ratios. *See* Tuning
Rate
audio (*see* Audio rate)
control (*see* Control rate)
sample (*see* Sample rate)
Recursion (recursive), 142, 154–155, 210, 221–224, 229, 593, 711–713, 786
Receiver, 16–17, 20, 27, 132–136, 148, 158, 708, 710–711, 714, 740–749, 802
Recording, 55, 93–94
References, 143
Reverb, 409, 417–420
Ringz. *See* Filter
RLPF (Resonant low pass filter). *See* Filter
Routine, 70, 82–88, 97–101, 115–116, 147, 175, 183, 199, 200, 328, 330, 336, 508, 514, 516, 602, 681, 788, 708

Sample and Hold, 51
Sample rate, 10, 32, 195, 298, 301, 304, 307, 310, 311–312, 409, 552, 559, 623, 764, 767, 803
Scale (collection of notes), 19, 37, 38–47, 167, 187, 191, 468–495, 540–541, 546, 573, 603–605, 608
microtonal (*see chapter 17*)
Scale (relative size), 49–65
time, 405–407, 411, 424–425, 443
scel. *See* Emacs
Scheduler, 627, 744, 798, 800, 808
Scheduling, 83–86, 112, 167, 195, 329, 330, 414,
constraints, 617, 626–629
sclang, 10, 83, 112, 125, 216, 280–293, 302–312, 445, 573–574, 578, 590–594, 620–622, 633–636, 722, 744–745, 748, 791, 795, 799
Scope, variable. *See* Variables, scope
Score, 87–91, 395. *See also chapter 18*
scsynth, 57–61, 66, 73. *See also chapter 26*
freeing, 196

Sequences (Sequencer), 44–47, 87–89, 136, 181–191, 214–218, 225, 233, 359, 540
SerialPort, 103, 114–117
Server, 10, 13, 17, 25–31, 54–62, 106, 112, 113–117, 305–306, 315, 378
 node, 208, 210
 synthesis, 278
 window, 10, 59, 61, 94, 267
Shaper, 63, 558
Shout Window, 239–243
Sidebands, 26–27
Slider, 10, 83, 119–123, 258–261, 317–337, 410, 598–608, 681, 704
Smalltalk, 1, 130, 176–177, 205, 235, 262, 265, 530–531, 719, 750, 751
Sonification, 2, 232, 248–251. *See also chapter 13*
Spatialization. *See chapter 14*
 3D audio, 343, 387, 388
Spectrum, 69, 80, 97, 117, 341, 346, 351, 359, 370, 396, 398, 410, 415, 435, 495
 diffusion, 399
 harmonic series, 64, 346, 405, 487
 inharmonic, 40
StartUp, 238, 524, 530, 796, 808–809
Streams, 87, 98–102, 188–189, 215–221, 224, 230, 470, 530. *See also* Patterns
String, 11–12, 17–18, 20, 29, 40, 48, 54
Subtractive Synthesis, 9
Switch (statements). *See* Flow control
SwingOSC. *See* GUI
Symbol, 18, 38, 131
streams, 221
Synth Definitions, 28, 30, 36–37, 118, 222, 490, 599, 620, 622. *See also chapter 6*
Synthesis, non-real-time. *See chapter 18*

Task, 35–37, 44–47, 82, 85–90
TaskProxy, 212–215, 252–261
Tempo clocks. *See* Clock, TempoClocks
Transcription. *See* Machine listening
Tuning. *See chapter 17*
 equal, 467–468
 just, 477, 493
 odd Limit, 477
 poly, 479–480
 ratios, 477
 tonality Diamond, 478
 unequal divisions, 475, 485–486

UGen (Unit Generator), 10. *See also chapter 2 and chapter 25*
 pseudo, 753–754
UI. *See* GUI
Unicode, 328, 678, 680
Unipolar, 50, 66
Unit Generator. *See* UGen
UNIX, 112–114, 282, 760–761
USB, 104, 295–303, 311–313, 603

Variables, 25–31, 46, 82, 132, 137–144
 class, 142

environment, 142
instance, 141, 158, 168, 200, 238, 524, 528, 725, 738
interpreter, 142, 180, 236
scope, 31, 155–156
versus references, 143
VBAP (Vector based amplitude panning), 390–391
View. *See* GUI
Voltage Control (VCO, VCF, VCA), 9, 23

Wave Field Synthesis, 387, 391–392, 400
Wave sets. *See* Granular synthesis
Wavetable, 162–163, 558, 714
Window
shout (*see* Shout window)

Code Index

This index contains language elements of SuperCollider. While most terms are used throughout the text, this index is limited to initial references, typically from the tutorial chapters. Note that this index is divided into two sections: messages and classes.

Messages

!=, 38
&&, 289, 481, 774, 784
% (mod, modulus), 19, 37–38
++, 47–48, 65
<<, 136, 137,
<=, 38, 165, 486, 774
<>, 169–170, 173, 213
==, 36, 38, 44
>=, 38, 170, 174

abs, 49, 456, 628–629
add, 11, 22–24, 48, 50
addAll, 136, 150
ampdb, 416
and (.and), 38
ar (.ar), 11–16
asArray, 202, 250, 488, 778
asAscii, 116
asCompileString, 167
asInteger, 45, 116, 416
asr, 365, 490–491
asStream, 88–89, 99–100, 182, 189–200
asString, 48, 136, 143, 159
at (accessing elements of a collection), 35, 45, 105, 110
audio, 74

background, 107, 269, 3221, 327–329
bufnum, 31, 385, 409–410

choose, 19, 37, 43–45, 99
class, 142, 172, 394
clear, 68, 83, 86, 91–92, 207, 213
clip, 368, 553, 592
clock, 202, 574–575, 577–578
close, 60, 65, 115–116, 145, 240–241
coin, 36–39, 458, 462
collect, 91, 143, 154
connect, 107–108, 110, 114, 329
control, 33–34, 523, 525, 674
copy, 192, 255–257, 258, 260

count, 159, 175
cpsmidi, 17, 67, 190
current, 176, 253–257, 259–261
curve, 324

dbamp, 70–71, 75–76
decorator, 96, 159, 258–260
def, 148, 155
default
  Server, 135, 151–152, 379
  TempoClock, 589
defer, 83, 176, 330, 410, 607
degreeToKey, 475
delta, 216, 632
device, 11, 105
disconnect, 114
discretize, 432, 439, 442, 444–445
do, 35–40, 44–46, 65
doOnce, 68
drop, 242, 244, 247, 356, 359
dump, 142, 148–149, 162, 237
dup, 17–18, 20, 77–78
duration, 398, 513, 528–529

embedInStream, 192–193, 199–200
env, 444, 511
envir, 254–255, 259–260, 629
error, 147, 250, 584, 591
even, 38
explin, 437, 621
exprange, 65, 214, 226–227

fadeTime, 213, 238, 445
fill
  Array, 38, 43–45, 52, 64
  Mix, 12, 42, 95, 598–599, 606
font, 145, 240–247
for, 153
forBy, 153
fork, 85, 140, 146
format, 579–583, 585
free, 26, 30, 46, 60, 65
freq, 436–437, 511, 515–525
front, 83, 86, 96, 107

gap, 410, 461, 463, 606
get, 356–357, 410, 417, 521
getn, 410, 803–806
global, 565, 568
gui, 119, 123, 544, 679

if, 36–39, 44–45, 91, 116
includes, 38
index, 121, 410, 606, 664

info, 125
init, 107, 110
insert, 40, 53–54
interpret, 136, 142, 248, 478
ir, 69–70, 251–252, 357, 360
isClosed, 240, 245, 260, 318
isEmpty, 591
isFloat, 38, 747
isInteger, 38
isKindOf, 161, 527
items, 259

key, 260, 325, 329
kr, 11–12, 16–17, 22–55

lag, 121, 325–328, 561, 669
latency, 202, 253, 443
lfo, 551, 518–522
linen, 32, 357–358, 442, 665
linexp, 116, 121–122, 190, 196
linlin, 116, 121–122
linrand, 399, 346, 445
load, 499, 508
loadCollection, 349, 355, 399

make, 201
map, 34, 121–122, 214
max, 51, 368, 395
midicps, 1719, 29–30, 43–44
midiratio, 437, 468, 469, 516
min, 51, 260, 350, 452
mod (%), 19, 25, 44–46, 492
mouseDownAction, 106, 332–333, 601–602
mouseUpAction, 602

newMsg, 502–509
next, 67–68, 87, 99
nextLine, 258–260
node, 258–260
normalizeSum, 335, 490–491
not, 329, 333
notEmpty, 416, 419
noteOff, 416, 419
noteOn, 108–110
numChannels, 31
numFrames, 31–32, 398, 663

odd, 38–39, 449
onClose, 92, 146, 153
options, 346, 410, 415
or, 39

path, 31
pause, 86, 93, 219

perc, 30, 44, 51–52, 82
permute, 19
phase, 783
play, 20, 22, 24, 25, 26
plot, 18, 20, 40, 349
poll, 48–49, 52, 65
pop, 207, 212, 416
post, 18, 35, 38, 45
postln, 18, 20, 35
pow, 25, 134–135, 155
put, 115, 134, 162, 201
putAll, 216, 260
pyramid, 150

quant, 224, 253, 256, 592

rand, 14, 17–20, 35
range, 34, 50–51, 70–71
ratios, 493
read, 90–93, 98, 116, 152
reciprocal, 65, 78–79, 95–96, 140, 147
record, 93
release, 70–79, 85–89
render, 183, 194
reset, 87, 99
reverse, 19, 115, 165
rotate, 19
round, 14, 17–19
rrand, 72, 74, 76, 162
run, 70, 96, 508, 516

sampleRate, 10, 384, 409, 416
sched, 68, 82–84
schedAbs, 628–629
scope, 20, 23, 34
scramble, 19, 40, 53–54, 165, 217–218
send, 228, 419, 599
sendBundle, 113, 202, 253, 438, 443
sendCollection, 357, 439, 442, 444–445
sendMsg, 112, 114, 170
set, 29–30, 65, 121–122, 141, 146–147
setn, 389–390, 631, 671, 674
setStarttime, 513
signalRange, 50
sine, 196, 251, 397, 425
slice, 250–251
softclip, 43, 368, 708
source, 212
sourceCode, 155
squared, 19, 132, 149–154
standardizePath, 498–508
start, 37, 44, 46, 87–89
startTime, 513–516

states, 86, 91–92, 96
stop, 46, 86–89, 93
stream, 100, 472, 476, 485, 487, 491
string, 150, 176, 240–244
stringColor, 241, 246
sum, 60, 64, 70, 118, 150, 212
sum3rand, 775

tempo, 83, 87, 202
trace, 104, 108, 113

uid, 110

value, 84, 88, 89, 96120–123
valueArray, 171
valueEnvir, 167
view, 96, 150, 159, 162
visible, 318, 321, 327

wait, 35–37, 44, 46–47
while, 153
window, 60, 65, 239–245, 326
wrapAt, 36–40, 44, 46
wrapPut, 45–46
write, 498

xrand, 153
Classes

Allpass (N, L, C), 35, 53, 383
AppClock, 83, 329, 602
Array, 19, 38, 40–47

BeatTrack, 408, 413–414
BiPanB2, 392
Blip, 20, 23–26, 44–47
BPF, 210, 414, 452
BrownNoise, 174, 214
BufAllpass (N, L, C), 383–384
BufChannels, 775–777
BufComb (N, L, C), 383–384
BufDelay (N, L, C), 383–384, 395
BufDur, 444–445, 451, 665
Buffer, 31, 33–35
BufFrames, 196, 384–385, 560–568
BufRateScale, 90–92, 98, 196
BufRd, 558–569, 775
BufSampleRate, 452, 507–508
Bus, 33–34, 74

Char, 162, 168, 328
Clip, 368, 524, 553
CmdPeriod, 68, 607
Comb (N, L, C), 16, 49, 77–78, 77–78, 383–383
Compander, 63–65

Convolution, 385–389

Decay, 77–78, 362.
Delay (N, L, C), 65, 383–384
DetectSilence, 44, 362, 777
Dialog, 97
Dictionary, 140, 182, 200
Dust, 12, 26, 43, 52, 64

EnvGen, 30, 32, 44, 51–52
Environment, 141–143, 166
Event, 163, 167, 201–202
EventStreamPlayer, 100–101, 197–200
ExpRand, 12, 14–15, 19, 52, 54

FFT, 309, 397, 399
FlowLayout, 159, 258, 260, 478
Free, 798–802, 805, 808
FSinOsc, 98, 252, 425
Function, 61

GrayNoise, 135
GUI, 599

Harmonics, 163
HPF, 349, 414, 556, 759

Impulse, 23, 25–37, 32, 51, 65
In, 33–34, 74, 105
IRand, 70

KeyState, 106
KeyTrack, 408, 415
Klank, 43, 53

Lag (Lag2), 69–70, 209–210, 669
Latch, 51
LeakDC, 368, 552–561
LFClipNoise, 32–33, 209–210, 214
LFDNoise (0, 1, 3), 69–70, 141, 558–564
LFNoise (0, 1, 2), 11–12, 16–17, 20–23, 32–24
LFPulse, 34, 49–51, 53
LFSaw, 16, 21, 49, 51, 56
LFTri, 26, 65, 146, 214
Limiter, 64–65
Line, 23–27, 32, 56, 77
Linen, 23–26, 90–92, 180, 197
LinExp, 116121–122, 190, 196
LinPan2, 381
LinXFade2, 381
LPF, 414, 553, 556–557, 561

MIDIIn, 107–108, 111, 329–330, 419
MIDIOut, 110
Mix, 12–13, 41–43

MouseX (MouseY), 22–23, 25, 28, 56, 118
MultiSliderView, 319, 410, 606–607

Ndef, 212
Nil, 84, 86–87, 144, 162
Node, 170, 185
NodeProxy, 210, 212
NRand, 69

OffsetOut, 82, 180, 195–196, 214, 226–227, 251–252
Out, 28–35

Pan2, 12, 30, 34–35, 42–44
Pan4, 381
PanAz, 357, 381–383
PanB (2), 63, 73, 392
Patterns
  Pbind, 100–101, 162, 182–183, 189–196
  Pbrown, 101, 189, 191, 436, 445, 452–453
  Pdef, 101, 212, 217–224, 229, 592
  Pfunc, 189, 192, 208, 243–244, 247–248
  Pkey, 190–192, 398, 437
  Pmono, 189
  Ppar, 189–192, 220–221
  Pproto, 203
  Prand, 99–100, 153, 162, 189–191
  Prout, 189–194
  Pseq, 87–89, 99–101, 142, 162
  Pseries, 223, 363, 457–462
  Pshuf, 219, 428, 485, 487, 488, 547
  Pspawner, 183, 222
  Pstep, 142, 189–194, 542–544
  Pstutter, 189, 436
  Psym, 221
  Ptpar, 193–194, 538, 548
  Ptuple, 218, 221
  Pwhite, 189–190, 220–221, 226–228
  Pxrand, 87–88, 99–101, 437
PinkNoise, 77–78, 209, 322–324, 357, 368, 391, 399
PlayBuf, 31–34, 63, 90–93, 98
PMOsc, 22–23, 27–30, 50–51, 56
PopUpMenu, 259, 319
PriorityQueue, 798–808
ProxyMixer, 226
ProxySpace, 207–212, 226–229, 236
PV_Diffuser, 396
PV_HainsworthFoote (PV_JensenAndersen), 408, 411
PV_MagMul, 399, 781

QuadN, 359

Ramp, 369, 504, 552, 620
Rand, 17, 19, 75
RandID, 69

RandSeed, 14, 70, 504, 512, 514, 517
RangeSlider, 261, 319, 321, 326
RecordBuf, 63, 770, 776, 778–780
Rect, 86, 91–92, 96, 107, 131, 138
ReplaceOut, 62, 74
Resonz, 63, 70, 93, 95–96, 106, 135
Ringz, 226–227, 362, 625, 630
RLPF, 12, 710–711, 714–716
Rotate2, 381, 392
Routine, 84, 86, 200, 330, 336–337, 602
RunningSum, 117–118, 407

SampleRate, 10, 194, 357, 360, 384, 409
Saw, 325, 546, 565
SelectX, 383
SelectXFocus, 383
SendReply, 66, 118, 406, 647, 803
SendTrig, 65, 67–68, 114, 118, 406, 411–412414, 777
Server, 60, 132, 141, 145
ServerOptions, 31, 61, 73, 500, 503, 505, 508–509, 519, 526, 529
SinOsc, 11–13, 16–17, 21–29
Slider2D, 319
SoundFile, 194
SoundIn, 66–67, 118, 323, 351
Splay, 639, 680
SplayAz, 381, 383
StaticText, 175, 259, 319, 326
StereoConvolution2L, 385–386, 389
SystemClock, 68, 82–84, 329, 509, 607

Tdef, 212–219, 224–229
TempoClock, 83–84, 86–87, 185
TGrains, 32, 64, 397, 438, 441
TIRand, 69–70
TRand, 23–27, 69