# 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.

```
// causes an "operation cannot be called from this Process" error
SystemClock.sched(1, {SCWindow.new.front});
// defer reschedules GUI code on the AppClock, so this works
SystemClock.sched(1, {{ SCWindow.new.front}.defer});
```

GUI, by the way, is short for Graphical User Interface and refers to things such as windows, buttons, and sliders. This topic is covered in detail in chapters 9 and 10, 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 nitty-gritty 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.

```
(
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.

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.

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

Figure 3.2 is 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

```
(
t = Task({
    loop({   // loop the whole thing
        3.do({   // do this 3 times
            x.release(0.1);
            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;// fermata
        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.view.decorator = FlowLayout(w.view.bounds);
Button.new(w, Rect(0, 0, 100, 20)).states_([["Play/Resume", Color.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_([["Finish", Color.black, Color.clear]])
    .action_({
        t.stop;
        x.release(0.1);
        w.close;
    });
)
```

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

that a Routine does, but is meant to be played only with a Clock. A Task provides some handy advantages, such as the ability to pause. As well, it prevents you from accidentally calling play twice. Try playing with the various buttons and see what happens.

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.

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 2 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 great 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.

Patterns are like templates for producing Streams of values. In order to use a Pattern, it must be converted into 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` messages, 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 1 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

```
(
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.

```
(// 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.release(2);
        x = Synth(\default, [freq: p.value.midicps]);
        q.value.wait;
    });
});
t.start;
)
t.stop; x.release(2);
```

**Figure 3.4**
Using Patterns within a Task.

`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, and can be useful for anything from generic compositions to algorithmically variable compositions.

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 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, 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. In any case, if you find over time that you prefer to work in "messaging style" rather than "object style," you may find `Score` useful. Figure 3.6 provides a short example. Score also provides some handy functionality for non-real-time synthesis (see chapter 18).

```
(
p = 64; // a constant note
q = Pseq([1, 2, 0.5], inf).asStream; // ordered sequence of durations
t = Task({
    loop({
        x.release(2);
        x = Synth(\default, [freq: p.value.midicps]);
        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.

```
(
SynthDef("ScoreSine",{ arg freq = 440;
Out.ar(0,
    SinOsc.ar(freq, 0, 0.2) * Line.kr(1, 0, 0.5, doneAction: 2)
)
}).add;
x = [
// args for s_new are synthdef, nodeID, addAction, targetID, synth 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": Score.

```
(
// 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 sounds/ folder into buffers
~someSounds = "sounds/*".pathMatch.collect{ |path |  Buffer.read(s, path)};
)
// now here's the score, so to speak
// execute these one line at a time
~nowPlaying = Synth("playbuf", [buf: ~someSounds[0]]);
~nowPlaying.release; ~nowPlaying = Synth("playbuf", [buf: ~someSounds[1]]);
~nowPlaying.release; ~nowPlaying = Synth("playbuf", [buf: ~someSounds[2]]);
~nowPlaying.release;
// free the buffer memory
~someSoundsBuffered.do(_.free);
```

**Figure 3.7**
Executing one line at a time.

### 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 "tape" part, is easily achieved in SuperCollider. There are many approaches to the construction of cue players. These range 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 ask is whether to play the sounds from RAM or stream them from hard disk. The former is convenient for short files, and the latter for substantial 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, "sounds/a11wlk01.wav"); //load a 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

```
(
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 = "sounds/*".pathMatch.collect{ |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.view.decorator = FlowLayout(w.view.bounds);
//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", Color.black,
Color.clear]]).action_({ n=0; ~nowPlaying.release; });
// free the buffers when the window is closed
w.onClose = { ~someSounds.do(_.free); };
)
```

**Figure 3.8**
Playing cues with a simple GUI.

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.

The middle 2 lines of the latter section of figure 3.7 consist of 2 statements, and thus do 2 things when you press the enter key to execute. You can of course have lines of many statements, which can all be executed at once. (Lines are separated by carriage returns; statements, by semicolons.)

The "1 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. Figure 3.8 is a simple example with a GUI.

SC also allows for streaming files in from disk using the `DiskIn` and `VDiskIn` UGens (the latter allows for variable-speed streaming). There are also a number of

third-party extension classes that do things such as automating the required housekeeping (e.g., Fredrik Olofsson's `RedDiskInSampler`).

The previous examples deal with mono files. For multichannel files (stereo being the most common case) it is simplest to deal with interleaved files.[2] Sometimes, however, you may need to deal with multiple mono cues. Figure 3.9 shows how to sort them based on a folder containing subfolders of mono channels.

## 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, for documenting a performance, or for 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. Often, this is the simplest way to capture your sounds. (See figure 3.10.)

After executing this, you should have a sound file in SC's recordings folder (see the doc for platform-specific locations) labeled with the date and time SC began recording: SC_YYMMDD_HHMMSS.aif. `Server` also provides handy buttons on the Server window (appearance or availability varies by platform) to prepare, stop, and start recording. On OSX it may look like this, or similar (see figure 3.11).

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 3 methods must be called before `prepareForRecord`. A common case is to change the sample format; the default is to

```
// gather all your folder paths
//this will path match each folder in the collection, i.e. we will have a collection
of collections of paths

~groupOfindivCueFolders = "sounds/*".pathMatch.collect{ | item |
(item.asSymbol++"*").pathMatch };

Post << ~groupOfindivCueFolders;  //see them all !

//check how many cues you will have in the end
~groupOfindivCueFolders.size;

//automate the buffering process for all cues:
~bufferedCues = ~groupOfindivCueFolders.collect{|item, i| item.collect{| path |
Buffer.read(s,  path)}}; //now all our cue files are sitting in their buffers !

~bufferedCues[0];  //here is cue 1

// see it in the post window:
Post << ~bufferedCues[0];

// play them all in a Group, using our previous synthdef
// we use bind here to ensure they start simultaneously
(
s.bind({
    ~nowPlaying = Group.new(s); // a group to put all the channel synths in
    ~bufferedCues[0].do({|cue| Synth("playbuf", [buf: cue], ~nowPlaying)})
});
)
// fade them out together by sending a release message to the group
~nowPlaying.release;
```

**Figure 3.9**
Gathering up files for multichannel cues.

```
s.boot; // make sure the server is running
(   // first evaluate this section
b = Buffer.read(s, "sounds/a11wlk01.wav"); // 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); // 'sticky' 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.

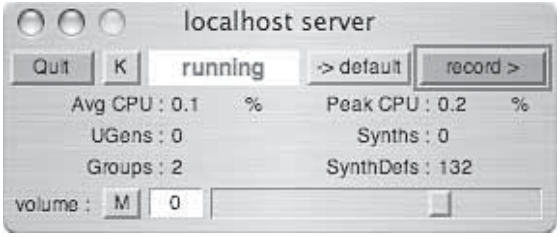


**Figure 3.11**
A screen shot of a Server window.

record as 32-bit floating-point values. This has the advantage of tremendous dynamic range, which means 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, of course, 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).

### 3.3.3 Gestures

For a long time, electroacoustic and electronic composition has been a rather "manual" process. This may account for the computer's 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 may in fact be replacing 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 "tape" 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 1 sound with the

```
(
f = 300;
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.

spectrum of another. The shape of 1 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 on, 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 dialogue box and can select any sound(s) on our hard disk:

```
( //you will be able to add multiple sound files; just shift click when
selecting!
var file, soundPath;
~buffers = List[];
Dialog.getPaths({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);); })
});
)
```

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 panning
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 more low-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, \bufnum, ~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 1 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, \bufnum, ~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.

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, 8], 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 1 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 routine for creating a ritardando stutter with panning, you must have
run the code in fig 3.9 so that this routine may find some sounds already loaded
into buffers, you can change the index of ~bufferedCues 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,  ~bufferedCues[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.

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).play;
```

It's also possible to substitute Event Streams as they play. When you call `play` on a `Pattern`, it returns an `EventStreamPlayer`, which actually 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, \rate,
Pxrand([1/2, 1, 2/3, 4], inf), \rel, 0.9).asStream; //substitute 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, 1.1707, 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], inf), \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 so it will
stop!
```

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

```
~gest1 = Pdef(\a, Pbind(\instrument, \samplePlayer, \dur, ~rhythm1, \rel,
1.9, \pSpeed, 0.5) );
~gest1.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`,

```
~gest1 = Pdef(\a, Pbind(\instrument, \samplePlayer, \dur, ~rhythm1, \rel,
1.9, \pSpeed, 0.5) );
```

we can hear that the new `~rhythm1` has taken the place of the previous one. Notice that it played immediately, without the need for executing `~gest1.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:

```
~gest1 = Pdef( \a , Pbind(\instrument, \samplePlayer, \att, 0.5, \rel, 3,
\lev, {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 1 `SynthDef`, we can even modulate the `\instrument` by getting it to choose among several different options. Once we have a set of gestures 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 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 re-create them yourself using SuperCollider. Below we refer you to some interesting texts that may enhance your creative investigations.

### Further Reading

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: 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 M. V. Mathews and J. R. Pierce, eds., *Current Directions in Computer Music Research,* pp. 291–396. Cambridge, MA: MIT Press.

Loy, G., and Abbott, C. 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.

## 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 convenient interleaving and deinterleaving of audio files.

3. A geometric series is a series with a constant ratio between successive terms.